ULISSE s.r.l.
Servizio Distribuzione Apple
Via Pacini, 22
20131 Milano
Tel. (02) 2367783-2367406

su licenza esclusiva
Apple Computer S.p.A.

Macintosh Programmer's Workshop Assembler Reference

Version 1.0

MPW01

Contents

Figures and Tables xi

Preface **About This Manual xv**
What This Manual Contains xv
Other Reference Materials xvii
Notation Conventions xvii
Computer Face xviii
Italic xviii
Fields xix
Delimiter Symbols xix
Braces xx
Brackets xx
Ellipses xx
Underlining xxi

Chapter 1 **About the Assembler 1**
General Characteristics 2
Overview of the Assembly Process 3
Assembly Files 4
Programming for the Macintosh 5
Macintosh Libraries 5

Chapter 2 **Coding Conventions 7**
Source Text Structure 8
Scope of Definitions 10
Imported and Exported Objects 11
@-Labels 12
Summary 12
Segmentation 13
Machine Instruction Syntax 14
The Label Field 14
The Operation Field 15

The Operand Field 17
Comments 18
Symbols 19
Identifiers 19
Numeric Constants 20
Strings 21
Expressions 22
Evaluation of Expressions 24
Absolute and Relocatable Expressions 25
Absolute Expressions 25
Relocatable Expressions 26

Chapter 3 Address Syntax 27

Addressing Modes 28
Ambiguities and Optimizations 31
Forward-Reference Addressing 33
Registers 33
Special Address Formats 35
MC68xxx Instructions 35
MOVEM: Multiple Moves 35
MC68020 Instructions 36
MULS and MULU: Signed and Unsigned Multiplication 36
DIVS and DIVU: Signed and Unsigned Division 36
TDIVS and TDIVU: Truncated Signed and Unsigned Division 36
PACK and UNPK: Packing and Unpacking 37
CAS and CAS2: Comparing and Swapping 37
Bit Field Instructions 38
Tcc and TPcc: Trap on Condition 38
MC68881 Instructions 38
FMOVEM with Explicit Register Lists 38
FMOVE with Packed BCD Data 39
FSINCOS: Simultaneous Sine and Cosine 39
FTcc and FTPcc: Floating-Point Trap on Condition 40
FTEST: Test Operand and Set Floating-Point Condition Codes 40
MC68851 Instructions 40
Literals 42

Macintosh Programmer's Workshop
Assembler Reference

October 3, 1986

Apple Technical Publications

This document contains preliminary information. It does not include

- *final editorial corrections*
- *final art work*
- *an index.*

It may not include final technical changes.

Apple Computer, Inc.

This manual and the software described in it are copyrighted, with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or part, without written consent of Apple, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies may not be made for this purpose.

© Apple Computer, Inc., 1986
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010

Apple, the Apple logo, LaserWriter , and Lisa are registered trademarks of Apple Computer, Inc.

Macintosh is a trademark of Apple Computer, Inc.

Simultaneously published in the United States and Canada.

Chapter 4 Assembler Directives 45

Directive Formats 48
Code and Data Module Definitions 48
PROC and ENDPROC: Define Procedure Code Module 49
FUNC and ENDFUNC: Define Function Code Module 50
MAIN and ENDMAIN: Define Main Program Code Module 50
RECORD and ENDR: Define a Data Module 51
INCREMENT and DECREMENT 52
MAIN 53
CODE and DATA: Switch Between Code and Data 54
END: End the Assembly 54
Symbol Definitions 55
EQU and SET: Name Constants and Registers 55
REG and FREG: Name Register List 56
OPWORD: Name Machine Instruction 58
Data Definitions 59
DC and DCB: Define Constants 59
DS: Define Storage Area 61
Template Definitions 62
RECORD and ENDR: Define Template 63
Using Templates as Data Types 67
WITH and ENDWITH: Supply RECORD Name Qualification 68
Linker and Scope Controls 70
EXPORT and ENTRY: Expand Scope of Entry Points 70
IMPORT: Identify External Entry Points 72
CODEREFS and DATAREFS: Control Name Linking 74
Code-to-Code References 75
Code-to-Data References 76
Data-to-Code References 76
Data-to-Data References 76
SEG: Define Current Segment 77
COMMENT: Place Comment in Object File 78
Assembly Options 79
MACHINE: Specify Target Machine 79
MC68881: Assemble MC68881 Coprocessor Instructions 79
MC68851: Assemble MC68851 Coprocessor Instructions 80
STRING: Specify String Format 81
BRANCH and FORWARD: Resolve Forward Branches 81
OPT: Specify Level of Code Optimization 82
CASE: Specify Treatment of Lowercase Letters 83
Writing Register Names 84
BLANKS: Control Acceptance of Blanks in the Operand Field 84
Location-Counter Controls 85

ALIGN: Align Location Counter 85
ORG: Set Location Counter 86
File Controls 87
File search rules 88
INCLUDE: Take Source Text from Another File 88
DUMP and LOAD: Write and Read Symbol Table Files 89
ERRLOG: Specify Error Log File 90
Listing Controls 90
PAGESIZE: Specify Listing Page Size 91
TITLE: Specify Title Line for Listing 91
PRINT: Control Listing Information 92
EJECT: Start New Listing Page 94
SPACE: Insert Blank Line in Listing 94

Chapter 5 Defining and Using Macros 95

Macro Expansion 97
Scope of Macro Symbols 97
Defining Macros 98
MACRO and ENDM or MEND: Delimit a Macro 99
The Prototype Statement 99
The Macro Body 100
Macro Comments 101
Symbolic Parameters 102
Concatenating Symbolic Parameters 103
Calling Macros 104
Macro call labels 105
Operand Syntax 105
Paired single quotes 106
Paired Parentheses and Brackets 106
Ampersands 107
Commas 107
Blanks (Spaces and Tabs) 107
Backquotes 107
@-Labels 108
Omitted or Extra Operands 108
Operand Sublists 109
Accessing Sublist Elements 109
Parameter Types and Default Values 110
Nesting Macros 111
Keyword Macros 112
Defining Keyword Macros 113
Calling Keyword Macros 113
Mixed-Mode Macros 115

Chapter 6 Macro Variables and Functions 117

SET Variables 118
SET Variables and Symbolic Parameters 120
LCLA, LCLC, GBLA, and GBLC: Define SET Variables 121
SETA and Integer Expressions 122
&ABS: Return Absolute Value 123
&EVAL: Evaluate Contents of String 123
&ISINT: Test String for Integer Content 124
&LEN: Measure String Length 124
&LEX: Parse String Lexically 124
&LIST: Divide String Into List 126
&MAX: Find Maximum in Integer List 127
&MIN: Find Minimum in Integer List 127
&NBR: Count Sublist Elements 128
&ORD: Return integer value 128
&POS: Find Position of Substring in String 129
&SCANNEQ and &SCANNE: Scan String 129
&STRTOINT or &S2I: Convert String to Integer 130
Symbol Table Functions 130
&NEWSYMTBL: Create New Symbol Table 131
&ENTERSYM: Enter or Update Symbol in Table 131
&FINDSYM: Find Symbol in Table 132
&DELSYMTBL: Delete Symbol Table 133
SETC and String Expressions 133
Accessing Substrings of String Variables 134
&CHR: Convert Integer to Character 135
&CONCAT: Concatenate Strings 135
&DEFAULT: Return String Value or Default 135
&GETENV: Return MPW Environment Value 135
&INTTOSTR or &I2S: Convert Integer to String 136
&LOWCASE or &LC: Convert String to Lowercase 136
&SETTING: Return Directive Setting 136
&SUBSTR: Return Substring of String 137
&TRIM: Trim Spaces and Tabs from String 138
&TYPE: Determine Identifier Type 138
&UPCASE or &UC: Convert String to Uppercase 140
SET Array Variables 140
Defining SET Array Variables 141
Using SET Array Variables 141
Accessing Substrings in SET Array String Elements 142
Assembler System Variables 143
&SYSNDX or &SYSINDEX: Macro Call Index 143
&SYSLIST or &SYSLST: Macro Operand List 144
&SYSSEG: Current Segment Identifier 145
&SYSMOD: Current Module Identifier 145

&SYSDATE: Current Date 145
&SYSTIME: Current Time 145
&SYSTOKEN and &SYSTOKSTR: Values Set by &LEX 146
&SYSVALUE and &SYSFLAGS: Values Set by &FINDSYM 146
&SYSLOCAL and &SYSGLOBAL: System symbol table IDs 146

Chapter 7 Macro and Conditional-Assembly Directives 147

Boolean Control Expressions 148
Comparing Two Integer Expressions 149
Comparing Two String Expressions 149
Comparing Integer and String Expressions 149
GOTO, IF...GOTO, and Macro Labels: Branching 150
IF, ELSEIF, ELSE, and ENDIF: Conditional Assembly 151
WHILE and ENDWHILE: Looping 152
ACTR: Limit Looping 153
EXITM: Exit Macro 154
WRITE and WRITELN: Write to diagnostic output file 154
AERROR: Error Generation 154
ANOP: Assembler NOP 155

Appendix A Generic Instruction Formats 157

Appendix B Syntax Diagrams 161

Assembly-Language Addresses 161
Addressing Modes 161
Address Optimizations 162
Special Address Formats 163
MC68000 Instructions 163
MOVEM: Multiple Moves 163
MC68020 Instructions 163
MULS and MULU: Signed and Unsigned Multiplication 163
DIVS and DIVU: Signed and Unsigned Division 163
TDIVS and TDIVU: Truncated Signed and Unsigned Division 164
PACK and UNPK: Packing and Unpacking 164
CAS and CAS2: Comparing and Swapping 164
Bit Field Instructions 164
Tcc and TPcc: Trap on Condition 165
MC68881 Instructions 165
FMOVEM with Explicit Register Lists 165
FMOVE with Packed BCD Data 165
FSINCOS: Simultaneous Sine and Cosine 165

FTcc and FTPcc: Floating-Point Trap on Condition 165
FTEST: Test Operand and Set Floating-Point Condition Codes 166
MC68851 Instructions 166
Literals 167
General Assembly Directives 167
Macro and SET Variable Directives 171
SET Variable Functions 172

Appendix C Assembly Listing Format 175

Appendix D Error and Warning Messages 179

Appendix E Other Assemblers 187

Syntax Comparison 187
Writing Identifiers 187
Writing Numbers 188
Writing Strings 189
Defining Modules 189
Communicating Between Modules 190
Writing Expressions 190
Location-Counter Reference 191
Addressing Features 192
Writing Macros 192
Conversion Programs 193
TLA Conversion 193
Limitations on TLA Conversion 196
Using TLACvt 196
MDS Conversion 197
MDS Conversion Strategy 200
Limitations on MDS Conversion 202
Using MDSCvt 203

Appendix F Macintosh Character Set 205

Appendix G Instruction Sets 207

Instruction Evaluation 208
Listing Conventions 208
Opcode 209
Operands 209
Opcode word 211

cp-type 211
Group 211
Flags 211
Range 212
Equivalent 213
Condition Codes 213
Instruction Set Listings 216

Appendix H Assembler Command Syntax 237

Syntax 237
Description 237
Input 237
Output 237
Diagnostics 237
Status 238
Options 238
Example 241

Appendix I Programming for the Shell Environment 243

General requirements 243
Importing the routines 244
Calling conventions 244
The RT_Init function 245
Files to link with 246
Shell Environment routines 246
Accessing MPW command-line parameters 246
The ArgC variable 246
How Shell commands are stored and accessed 247
Shell utility routines 247
The StandAlone variable 247
The getenv function 247
The IEGetEnv Function 248
The onexit Procedure 248
The exit and _exit procedures 248
The _RTExit procedure 249
Shell I/O routines 249
Error information 249
The faccess Function 251
The open Procedure 252
The close Function 253
The read Function 254
The write Function 254
The ioctl Function 254
The lseek Function 256

The fcntl function 256
Shell Signal Handling Routines 257
The sigset Function 258
The sighold Procedure 258
The sigrelease Procedure 258
The sigpause Procedure 259
MPW cursor control routines 259
The Show_Cursor Procedure 259
The Hide_Cursor Procedure 260
The RotateCursor Procedure 260
The SpinCursor Procedure 261
Error message file management routines 261
The InitErrMgr Procedure 261
The GetSysErrText Procedure 261

Appendix J The Graf3D Library 263

How to Use Graf3D 264
Graf3D Data Types 264
Point3D 264
Point2D 265
XfMatrix 265
Port3DPtr 265
Graphics Routines 266
The InitGraf3D Procedure 266
The Open3DPort Procedure 266
The SetPort3D Procedure 267
The GetPort3D Procedure 267
The Move Procedures 267
The Line Procedures 267
The Clip3D Function 268
The Set Point Procedures 268
Setting Up the Camera 268
The ViewPort Procedure 268
The LookAt Procedure 269
The ViewAngle Procedure 269
The Transformation Matrix 269
The Identity Procedure 269
The Scale Procedure 269
The Translate Procedure 270
The Pitch Procedure 270
The Yaw Procedure 270
The Roll Procedure 270
The Skew Procedure 270
The TransForm Procedure 271

Appendix K Object Assembler Macros 273

InitObjects 273
ObjectDef 274
ObjectIntf and the IMPL Keyword 275
ObjectWith and EndObjectWith 277
ProcMethOf, FuncMethOf, and EndMethod 277
MethCall 278
Inherited 279
NewObject 279
MoveSelf 279

Appendix L Pascal and C Calling Conventions 281

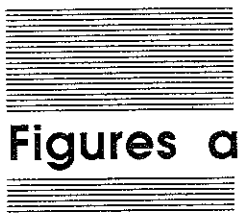
Pascal Calling Conventions 281
- Parameters 281
  - Real-Type Parameters 282
  - Structured-Type Parameters 282
- Function Results 284
- Register Conventions 285

C Calling Conventions 286
- Parameters 286
- Function Results 286
- Register conventions 286

Appendix M The MC68030 Processor 287

Assembler control 287
MC68020 statements you can use 287
MC68851 instructions you can use 288

Glossary 291



Figures and Tables

Chapter 1 About the Assembler

Table 1-1 Assembler Status Codes 4

Chapter 2 Coding Conventions

Figure 2-1 Source Text Structure 9
Table 2-1 Scope of Identifiers 13
Table 2-2 Data Size Qualifiers 16
Table 2-3 Operators 23

Chapter 3 Address Syntax

Table 3-1 Address Symbols 28
Table 3-2 Address Syntax Summary 29
Table 3-3 Effective Address Optimizations 32
Table 3-4 Registers 33
Table 3-5 Special MC68851 Operand Formats 40

Chapter 4 Assembler Directives

Table 4-1 DC and DCB Data Increments 59
Figure 4-1 Stack Frame Example 65
Table 4-2 Effects of CODEREFS and DATAREFS 77
Figure 4-2 Sample Template Format 87
Table 4-3 PRINT Directive Parameters 92

Chapter 6 Macro Variables and Functions

Table 6-1 Values Returned by &LEX 125
Table 6-2 &SETTING Values 137
Table 6-3 Assembler System Variables 143

Appendix A Generic Instruction Formats

Table A-1 Generic Instruction Conversions 158

Appendix C Assembly Listing Format

Figure C-1 Default Assembly Listing Format 176

Appendix D Error and Warning Messages

Table D-1 Error and Warning Messages 180
Table D-2 Assembler I/O Errors 185

Appendix E Other Assemblers 187

Table E-1 Identifier Syntax Rules 188
Table E-2 Number Syntax 189
Table E-3 String Syntax 189
Table E-4 Module Definition 190
Table E-5 Communication Directives 190
Table E-6 Allowable Operators 191
Table E-7 Addressing Features 192
Table E-8 Normal TLA Token Conversion 193
Table E-9 TLA Macro Token Conversions 194
Table E-10 TLA Directive Conversions 194
Table E-11 Normal MDS Token Conversion 198
Table E-12 MDS Macro Token Conversions 198
Table E-13 MDS Directive Conversions 199

Appendix G Instruction Sets

Table G-1 Instruction Operands 210
Table G-2 MC68xxx condition codes 213
Table G-3 MC68881 IEEE non-aware tests 214
Table G-4 MC68881 IEEE aware test 214
Table G-5 MC68881 miscellaneous tests 215
Table G-6 MC68851 PMMU condition codes 215
Table G-7 MC68000, 68010, and 68020 Instructions 216
Table G-8 MC68881 Instructions 226
Table G-9 MC68851 Instructions 233

Appendix I Programming for the Shell Environment

Table I-1 Shell I/O Errors 250
Table I-2 Predeclared file descriptors 253
Table I-3 Cursor kinds 260

Appendix L Pascal and C Calling Conventions

Table L-1 Parameter Passing Conventions 283
Table L-2 Function-Result Passing Conventions 284

Appendix M The MC68030 Processor

Table M-1 MC68851 registers in the MC68030 288
Table M-2 MC68851 instructions valid for the MC68030 288

Preface

About this manual

This manual tells you how to prepare source files to be assembled by the Macintosh™ Programmer's Workshop Assembler (also called MPW Assembler), a part of the Macintosh Programmer's Workshop.

This manual assumes that you are generally familiar with assembly-language programming. It also assumes that you understand and are able to write the symbolic assembly language for the Motorola MC68xxx instructions you want to use.

What this manual contains

This manual is divided into 7 chapters and 13 appendixes. Here is a summary of the information it contains:

- Chapter 1, "About the Assembler." This chapter lists some characteristics of the Macintosh Workshop Assembler and describes its general mode of operation. It also includes a summary of file-naming conventions.
- Chapter 2, "Coding Conventions." This chapter discusses the overall structure of MPW assembly-language source text. It includes information about statement and directive formats, symbol formation, and the evaluation of expressions.
- Chapter 3, "Address Syntax." This chapter describes the ways you can address the Macintosh memory and gives the syntax rules for writing addresses in your source text.
- Chapter 4, "Assembler Directives." This chapter provides detailed instructions for using most of the MPW Assembler directives, grouped by the kinds of tasks the macros perform. Macro-expansion directives are covered in Chapter 7.
- Chapter 5, "Defining and Using Macros." This chapter tells you how to define and call macros in your source text.

- Chapter 6, "Macro Variables and Functions." This chapter tells you how to use SET variables to program the expansion of your macros.
- Chapter 7, "Macro and Conditional-Assembly Directives." This chapter describes the directives of the MPW Assembler macro language.
- Appendix A, "Generic Instruction Formats." The MPW Assembler accepts certain assembly-language statements in generic form, converting them to specific instructions. This appendix gives you the rules for writing these generic forms.
- Appendix B, "Syntax Diagrams." This appendix contains copies of all the syntax diagrams used in this manual.
- Appendix C, "Assembly Listing Format." This appendix describes the way the Assembly listing is constructed.
- Appendix D, "Error and Warning Messages." This appendix lists all errors and warnings that can be reported by the Assembler.
- Appendix E, "Other Assemblers." This appendix compares the Macintosh Workshop Assembler with other assemblers available for the MC68000, and tells you how to use programs that translate source text from other forms.
- Appendix F, "Macintosh Character Set." This appendix lists the characters in the Macintosh character set and gives their numeric values.
- Appendix G, "Instruction Sets." This appendix lists the MC68000, MC68010, MC68020, MC68851, and MC68881 machine instructions and condition codes accepted by the Assembler.
- Appendix H, "Assembler Command Syntax." This appendix defines the syntax for writing the Assembler command line, including information about the Assembler options.
- Appendix I, "Using the Integrated Environment." This appendix tells you how to access the routines and variables of the MPW Shell environment.
- Appendix J, "The Graf3D Library." This appendix explains the three-dimensional drawing routines provided in the Graf3D library.
- Appendix K, "Object Assembler Macros." This appendix describes the macros provided for object-oriented programming in the MPW assembly language.
- Appendix L, "Pascal and C Calling Conventions." This appendix gives the assembly-language calling conventions for routines written in Pascal and C.
- Appendix M, "The MC68030 Processor." This appendix tells you how to write source text for the MC68030 processor.

At the end of this manual you will find a glossary and an index. Terms appearing in **boldface type** in the text are defined in the glossary.

Other reference materials

Before trying to write and assemble a Macintosh assembly-language program, you should read and understand the following books:

- Apple Computer *Macintosh Programmer's Workshop Reference.* A full description of how to use the Workshop's program preparation tools, including the Assembler.
- Motorola *M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual,* 5th edition. Prentice-Hall, 1986. The latest comprehensive guide to the MC68000 microprocessor.

In addition, you may find these books helpful:

- Apple Computer *Inside Macintosh.* Volumes I-III, Addison-Wesley, 1985. The complete story of the architecture and operation of the 128K and 512K Macintosh, including details on these ROM routines.
- Apple Computer *Inside Macintosh.* Volume IV, Addison-Wesley, 1986. Additional and updated material covering the Macintosh and the Macintosh Plus.
- Apple Computer *Apple Numerics Manual* Addison-Wesley, 1986. A description of the Standard Apple Numeric Environment and how it is used with the Macintosh computer.
- Motorola *MC68020 32-Bit Microprocessor User's Manual.* A guide to the MC68020 instructions and addressing modes.
- Motorola *MC68851 Paged Memory Management Unit User's Manual.* A guide to the MC68851 coprocessor, with details of all its instructions.
- Motorola *MC68881 Floating-Point Coprocessor User's Manual.* A guide to the MC68881 coprocessor, with details of all its instructions.

Notation conventions

The discussions in this manual include a number of syntax diagrams and source text examples, designed to help you understand exactly how to write source text structures. Syntax diagrams appear as appropriate in the text and are also gathered together in Appendix B. This section tells you how to interpret them.

Computer face

Anything printed in `COMPUTER FACE` is a sample of actual source text, as it might be processed by the Assembler. Where computer face occurs in syntax diagrams, it indicates fixed symbols of the MPW Assembly language. Such symbols, which must be written in the source text as they are written in the syntax diagram, are sometimes called **terminal symbols**.

❖ *Note:* To further distinguish them, directives and machine instructions are printed in capital letters. However, you do not need to use capitals in your source text.

Italic

Generic terms that designate information to be supplied by the programmer are printed in *italic*. They are sometimes called **nonterminal symbols.** Such terms may contain hyphens instead of spaces. The most common ones are

abs-expr	An absolute expression
arith-expr	A numeric expression
rel-expr	A relocatable expression
expr	An absolute or relocatable expression
string	A string constant
str-expr	A string expression
name	An identifier
label	An identifier used as a label
macro-label	A macro label
filename	A string expression representing a filename
rlist	A `MOVEM` register list
reg	Any MC68000, MC68010, MC68020, or MC68030 register

Nonterminal symbols not shown in this list are defined where they are used for specific statements.

Where a nonterminal symbol is repeated in a syntax diagram, the repetitions are sometimes distinguished by subscripts. For example:

abs-expr$_1$,*abs-expr*$_2$

You write two absolute expressions at this point, separated by a comma. The expressions may be the same or different.

In some syntax diagrams, the connective ::= is used to show the possible values for a nonterminal symbol. For example,

size ::= W | L

indicates that you may write either *W* or *L* for the nonterminal symbol *size*.

References to nonterminal symbols in explanatory text are printed in *italic*, so you won't confuse them with ordinary words or phrases.

Fields

Syntax diagrams distinguish the fields into which source text lines are divided by horizontal spacing. For example:

[*macro-label*] INCLUDE *filename*

The nonterminal symbol *macro-label*, enclosed in brackets, indicates that you may write an optional macro label in the first field. The terminal symbol INCLUDE indicates that you must write the word *include* in any combination of uppercase and lowercase in the second field. The nonterminal symbol *filename* indicates that you must write a file name in the third field. For further details about fields, see "Machine Instruction Syntax" in Chapter 2.

Delimiter symbols

You must write the following delimiter symbols in your source text exactly as they are shown in the syntax diagrams:

,	Comma
()	Parentheses
.	Period
*	Asterisk
=	Equal sign

Occasionally, required delimiters may look like part of the syntax diagram's punctuation. To prevent confusion in such cases, the required marks are enclosed in single quotation marks. For instance, the expression '{'origin'}' means that the word *origin*, enclosed in braces, is required.

Braces

Material within braces represents required items, one of which must be chosen. For instance,

```
{ALPHA }
{BETA  }
{GAMMA }
```

indicates that you must write either *alpha*, *beta*, or *gamma* at that point in your source text.

Notice that the alternatives are written on separate lines. Terms on separate lines always represent expressions with distinct meanings. In some instances, alternate choices have the same effect. Such alternatives that mean the same are separated by a vertical bar (|). For example:

```
{ON  | YES}
{OFF | NO }
```

Because all the choices in this example are enclosed in braces, you must choose one line or the other. If you choose the first line you may write either *on* or *yes*; if you choose the second line you may write either *off* or *no*.

Brackets

Material within brackets represents optional items. Braces within brackets signify that one of the alternatives must be chosen if the material is to be included at all. Here are some examples:

[*abs-expr*]	You may write an absolute expression at this point, or nothing.
INCR[EMENT]	You may write either *incr* or *increment*
`[{ENTRY }]` `[{EXPORT}]`	You need not write anything, but if you do, it must be either *entry* or *export*.

Ellipses

An ellipsis containing three dots (...) indicates repetition of the preceding material. If the material is enclosed in brackets, you don't have to write it at all; otherwise you must write it at least once. A comma before the ellipsis indicates that repetitions must be separated by commas. You may repeat the material indefinitely, subject to the general length limitation for that particular source text structure. Here are two examples:

abs-expr,... Write one or more absolute expressions separated by commas

[*abs-expr*],... Write nothing, or write one or more absolute expressions separated by commas

Occasionally this notation can be ambiguous, in which case a longer form is used. For example,

`-d[efine]` *name*[=*value*] [, *name*[=*value*]]...

indicates that you may write more *name* or *name=value* groups after the first one, with a comma preceding each one.

An ellipsis containing two dots (..) indicates a scalar range. For example, 0..127 means "0 and 127 and all the intervening numbers."

A sequence of three hyphens (- - -) in a syntax diagram indicates lines of source text not specified by the diagram.

Underlining

An underlined item indicates a default or preset value, which the Assembler will assume if you omit an optional parameter. For example:

You need not write anything; but if you write nothing, the Assembler will act as if you had written "entry"

Chapter 1

About the Assembler

The MPW Assembler is contained in a single file, named Asm. This chapter describes some of its general characteristics. For instructions on invoking the Assembler, including further information about the environment in which it runs, see the *Macintosh Programmer's Workshop Reference*. A summary of Assembler command syntax and options is given in Appendix H.

General characteristics

The Macintosh Workshop Assembler is a Macintosh program that reads your **source text** and creates a file of linkable MC68000 **object code.** It has the following principal features, which help you build powerful assembly-language programs:

- It supports all the instructions and addressing modes for the MC68000, MC68010, MC68020, and MC68030 microprocessors, the MC68851 Paged Memory Management Unit (PMMU), and the MC68881 Floating-Point coprocessor, in all combinations.
- It has powerful macro capabilities, which handle both positional and keyword macros. These capabilities resemble those of the macro facilities in the IBM 360/370 Assemblers.
- Its macro capabilities accept global and local variables (called SET variables) that allow macros to communicate with one another. SET variables may contain numbers, characters, strings, or arrays. You can use them with conditional and looping statements to control the generation of complex structures of object code.
- It gives you the choice of creating either a single object module or a series of separate object modules.
- It gives you full control over the generation of both code and data modules, including A5-relative data. You can share global data between your assembly-language routines and routines written in MPW Pascal and MPW C.
- It lets you specify the scope of all code and data definitions. You can make the objects they define accessible only within modules, within files, or between files.
- It lets you define templates that determine the mapping of data in memory. Their function is similar to that of Pascal records or C structures. You can use templates as data types in much the same way as you use record types in Pascal.
- It lets you store its global symbol tables in files and then use these files for new assemblies. This increases assembly speed and saves disk space.

- It can generate Pascal-formatted and C-formatted strings, as well as fixed-length strings.

Overview of the assembly process

The MPW Assembler processes your source text in two **passes**. The first pass reads the source text, defines and expands all macros, and defines all symbols. It determines the length of the object code and resolves forward references. It also creates the following symbol tables:

- a **global symbol table** for symbols defined outside code or data modules
- a **macro symbol table** for all macro symbols and definitions
- a **local symbol table** for each code or data module

These operations do not create any object code.

As it starts to read each code or data module on its first pass, the Assembler creates an internal file in Macintosh memory containing a translation of your source text into **postfix notation.** At the end of each module, the Assembler performs its second pass, converting the postfix file into object code. The Assembler appends this code to the growing object code file. After processing each module, it releases memory held for the internal postfix file and the local symbol table.

Thus the Assembler translates each module separately, releasing the memory used before the next module is started. Nevertheless, symbol tables, macro definitions, and the postfix file all compete for the Macintosh's memory. To permit you to assemble large programs with limited RAM space, the MPW Assembler lets the postfix file spill over onto disk. When this happens, the Assembler returns a warning message at the end of the assembly process and assembly time increases by about 25 percent. Hence once there is enough RAM space available for its basic operations (including maintaining all symbol tables), the only memory limitation on the assembly process is the availability of disk space.

If you are generating an assembly listing, the Assembler creates a **scratch file** on the disk, in addition to the listing file. It writes source text lines that occur outside modules directly to the listing file, during the first pass. It writes lines that occur inside modules (including conditionally assembled lines and macro expansions) to the scratch file during the first pass, and then from the scratch file to the listing file during the second pass. To ensure that they appear in the correct location in the listing, the Assembler generates additional postfix code. As a consequence, assemblies that create listing files generate more postfix code.

During assembly, the Assembler sends errors and warnings to the **diagnostic output file** (the active window, unless you specify otherwise). If you use the `-P` Assembler option, described in Appendix H, the Assembler also writes progress and summary information to the diagnostic output. Status codes that the Assembler may send to the MPW Shell are listed in Table 1-1.

Table 1-1
Assembler status codes

Code	Status
0	No errors detected in any files assembled
1	Assembler command line parameter or option errors detected
2	Assembly processing errors detected
3	Assembly terminated before completion

Assembly files

By convention, you add the suffix *.a* to your assembly-language source text files. Object code files created by the Assembler are normally named after your source text file with the suffix *.o* added. However, you can change the name that the Assembler assigns to your object file by using the **-O** Assembler option, described in Appendix H. If you tell the Assembler to create a listing file, it will be named after your source text file with the suffix *.lst* added.

For example, given an assembly-language source text file *name*`.a` the Assembler will create an object code file *name*`.a.o` and a listing file *name*`.a.lst` from it.

In addition to the Assembler itself, the MPW disks contain library files of useful routines, together with the corresponding files of assembly-language interface statements, macros, and equates that access them. Your program can use any of these files. Files whose names end in `.a` contain assembly-language statements that you can include in your source text. Files whose names end in `.o` must be linked to your assembly. They contain executable code called by the assembly-language files. Most of the available libraries and their interface files are described in the *Macintosh Programmer's Workshop Reference;* some are described in this book, in Appendixes I, J, and K.

Programming for the Macintosh

There are three kinds of programs you can write for execution on the Macintosh:

- applications
- tools—programs that run under the MPW Shell
- desk accessories and other drivers

General information about building and installing these kinds of programs is given in the *Macintosh Programmer's Workshop Reference*. The following notes are specifically applicable to assembly-language programs.

- If an application contains one or more data modules containing `DC` or `DCB` directives, it must be linked with the library Runtime.o, which contains the data initialization routine `_DataInit`. Its first executable statement must be a call (`JSR`) to the entry point `_DataInit`. This entry point must also be declared as `IMPORT`. After returning from `_DataInit`, your program may unload the segment `%A5Init` that contains it, by calling the Macintosh routine `UnloadSeg`.
- Routines you can use with tools that run under the MPW Integrated Environment are described in Appendix I.
- Assembly-language desk accessories may not declare any global data. They must be linked with the file DRVRRuntime.o, which contains the Main code module for all desk accessories.

Macintosh libraries

Inside Macintosh describes an extensive group of **Macintosh library routines**, also called **Operating System routines** and **Toolbox routines**. They perform jobs such as creating menus, windows, and dialog boxes, providing simple text editing, and accessing files and devices.

Many of the Macintosh library routines are implemented in the Macintosh 64K and 128K ROMs. You can call them from an assembly-language program by using machine instructions whose high-order four bits are %1010 (that is, whose opcodes begin with $A). Such machine instructions are called **unimplemented**. The Macintosh **trap dispatcher** determines which of the library routines to call by examining the rest of the opcode.

The opcodes for various Macintosh library routines are defined by `OPWORD` directives contained in assembly-language files in the MPW folder {AIncludes}. If you include the appropriate files in your assembly, you can call the routines they cover by writing the routine identifiers (such as `_Read`) as operations in instruction statements.

Certain Macintosh library routines are in library object files, instead of in ROM. They are flagged in *Inside Macintosh* with the notation [Not in ROM]. You call these routines with JSR instructions. If you use any of them in your program, you must link your assembly with the MPW file {Libraries}Interface.o, which contains their code.

Additional information about calling Macintosh operating system and toolbox routines from assembly-language programs is contained in the chapter "Using Assembly Language" in *Inside Macintosh,* Volume I, Chapter 4. The include files and library files supplied with MPW are described in the *Macintosh Programmer's Workshop Reference.*

Chapter 2

Coding Conventions

This chapter describes the syntax rules and overall form required for source text that is to be processed by the Macintosh Workshop Assembler.

Source text structure

The source text formats for most higher-level programming languages are similar in structure. They consist of related procedures and functions plus various forms of data. The programs that interpret them differ mainly in the ways they support relationships among the routines and data. The Macintosh Workshop assembly language, although not a higher-level language, includes many of the programming facilities found in higher-level languages.

In order to understand the structure of a Macintosh Workshop assembly-language source text, it is helpful to understand the various components that make up a linked and executable program. Therefore, this discussion begins by defining some terms that describe an executable Macintosh program, together with how they relate to an assembly source text and the environment in which the program is executed.

Each line of Macintosh Workshop Assembler source text is either a **machine instruction statement** or a **directive statement.** Machine instruction statements generate executable code, using MC68xxx instructions. Directive statements are commands to the Macintosh Assembler to perform certain operations during assembly. The syntax rules for writing machine instruction statements are given later in this chapter. The syntax rules for writing directive statements are given under "Directive Formats" at the beginning of Chapter 4.

Every executable assembly-language program is built from a collection of **object files.** Each object file corresponds to one assembly and is made up of a collection of code and data **modules.** Each module contains one contiguous piece of code or data. Data modules represent static data, because the data space is defined before the program begins and the data remains accessible during the entire execution of the program.

When you link a program, the Linker groups all the code modules together and makes a separate grouping of all the data modules. Thus a linked object file consists of two parts: a collection of code modules and a collection of data modules.

In the Macintosh, the Segment Loader takes the collection of data modules and loads them into an area called the **application globals area.** This area is just below the area pointed to by register A5, called the **application parameter area.** Thus when the Linker adjusts code references to data in the data modules, it does so by setting negative offsets relative to A5, the assumed base register for data access. References by code to other code are made by jumping indirectly through a structure called the **jump table,** which is also built by the Linker and loaded by the Segment Loader just above the application parameter area. So the jump table is accessed by positive offsets from A5. A map of all these memory areas is included in Figure 9 of the Memory Manager chapter of *Inside Macintosh.*

Figure 2-1
Source text structure

Source text:

[{ *directive statement* / *data module* / *code module* }] . . .

END

Data module:

name RECORD

[*directive statement*] . . .

[ENDR]

Code module:

name {PROC | FUNC | MAIN}

[{ *directive statement* / *machine instruction statement* / *data section* / *code section* }] . . .

[ENDP | ENDF | ENDMAIN]

Data section:

DATA

[*directive statement*]. . .

Code section:

CODE

[{ *directive statement* / *machine instruction statement* }] . . .

Figure 2-1 is a syntax diagram that covers the overall structure of an assembly-language source file. The modules and directive statements in such a source file may occur in any order, subject only to the scope rules given later in this section under "Scope of Definitions."

Each module starts with a **module directive** (PROC, FUNC, MAIN, or RECORD). Each module terminates at the start of the next module directive, at the matching END*x* (ENDP, ENDF, ENDMAIN, or ENDR), or at the end of the source text. The end of the source text is indicated by an END directive.

Code modules are always introduced explicitly by either a PROC, FUNC, or MAIN directive. There is no structural difference between PROC and FUNC directives; FUNC is used instead of PROC only for documentation purposes. MAIN is essentially the same as PROC or FUNC, but has the additional function of indicating that this module is the **main code module,** and that its first instruction is the execution starting point for the program. There can be only one main code module in a linked program.

There are three ways to declare data in an assembly-language source file:

- As a data module introduced by the RECORD directive. Such a module generally contains data-definition and storage-allocation statements. It may also contain initialized values. All the data defined between the RECORD directive and its matching ENDR (or the start of the next module) generates a single data module.
- As a data module corresponding to one data-definition statement. Before the first module, or between explicitly declared modules, you may write statements outside of explicitly declared modules define their own data modules, one corresponding to each statement.
- As data that is part of a code module. Although you use the PROC, FUNC, and MAIN directives to indicate the start of a code module, you may generate an associated data module inside the code module, using the CODE and DATA directives. CODE and DATA may be used only inside a code module (PROC, FUNC, or MAIN); they indicate a switch from code to data (DATA), and then back to code again (CODE). Hence they delimit sections of code or data within the code module. Code in the code sections is generated contiguously—the first byte of one code section immediately follows the last byte in the previous code section. Similarly, the data in the data sections is contiguous.

Scope of definitions

The **scope** of a definition is the area of source text in which the code or data object it defines is accessible—that is, the area in which the object can be accessed by code or data statements. Scope rules permit you to restrict the scope of definitions. This lets you allow communication among the various routines of your program, while at the same time making selected objects inaccessible to other routines. Selectivity of scope promotes structured programming and helps you avoid identifier conflicts.

Here are the scope rules:

- All code or data definitions in a source file have either **global** or **local** scope.
- Local definitions override global definitions.
- The scope of a global definition extends from the point at which it occurs in the source file to the end of that file. Global definitions include those declared outside of a code or data module as well as definitions of code and data module identifiers. All identifiers assigned to global objects must be unique within the assembly.
- The Assembler permits field identifiers within a data module (created by the directive RECORD) to be accessed as **qualified** identifiers. Qualified identifiers are written in the form *modname.fieldname,* where *modname* is the data module identifier and *fieldname* is a data-definition field identifier, as defined within the data module. Field identifiers accessed in this way have global scope.
- A definition is considered to have local scope if it occurs inside a code or data module. Local objects may be accessed only from within the module; you may use the object's identifier in different modules or outside the module without causing an identifier conflict.
- The global/local scope rules may be overridden with the ENTRY, EXPORT, and IMPORT directives described in Chapter 4.
- ENTRY forces specific identifiers to be global. An identifier that is to be declared as ENTRY must be so declared before it is defined. From that point on, the identifier follows the same rules as global identifiers. This means that ENTRY may be used to access identifiers forward in your source text, such as labels in PROC directives.
- Since all data objects outside of modules, as well as the module identifiers themselves, have global scope, they are implicitly declared as ENTRY by the Assembler. For documentation purposes, a module identifier may also be declared explicitly as ENTRY.

Imported and exported objects

Local or global code or data objects may be made accessible to source text files other than the file in which they are defined. Objects defined in a file, intended to be accessed outside it, are said to be **exported.** Objects accessed from outside the file in which they were defined are said to be **imported.** Thus an exported object in an object file can be imported into any number of other object files. You export and import objects by using the EXPORT and IMPORT directives.

Using EXPORT inside a code or data module declares specified local identifiers as exported. You must use EXPORT before defining the specified identifiers. The identifiers may then be accessed from other files that import those identifiers. Since an exported local identifier is made accessible outside of the module in which it is defined, EXPORT promotes local identifiers to global scope within the same source text file, just as ENTRY does. Module identifiers, code or data labels, global data definitions, and storage-allocation identifiers may be exported.

Once an object's identifier is declared as EXPORT, other source text files may import the identifier by using IMPORT. IMPORT declares specified identifiers as local or global, depending on where the IMPORT statement is used within the file. If IMPORT is used inside a module, then the identifiers are local to that module. Using IMPORT outside a module declares the identifiers as global to the file.

Since EXPORT identifiers are global to the file in which they are declared, the Assembler treats references to such identifiers from modules other than the one that actually defines the identifier as imports of those identifiers. For documentation purposes, however, you can always import such identifiers explicitly by using IMPORT in the same file.

@-labels

Label identifiers that begin with an at symbol @ are called **@-labels.** They have more limited scope than other labels and can't be used in directives or outside modules. Specifically, the scope of an @-label extends through the source text, in both directions, to the nearest label that doesn't begin with @. @-labels may be redefined, but not in the scope of another instance of the same @-label. All @-labels defined or used inside macros follow the same rules, but in addition their scope is limited to the body of the macro. @-labels passed as macro parameters retain the scope they had when the macro was called, with certain restrictions. For further information about passing @-labels to macros, see "Operand Syntax" in Chapter 5.

Summary

Table 2-1 summarizes the kinds of identifiers used in definitions of different scope.

Table 2-1
Scope of identifiers

Temporary scope (can be accessed only within a part of a module):
    @-labels (beginning with @)

Local scope (can be accessed only from within the module; overrides global declarations):
    All identifiers defined within a code or data module
    Identifiers imported by using `IMPORT` within a module

Global scope (can be accessed from the point of definition to the end of the file):
    Code and data module identifiers
    All identifiers defined outside of code and data modules
    Identifiers imported by using `IMPORT` outside of any module
    Identifiers declared as `ENTRY`
    Qualified data module identifiers

Identifiers accessible between files:
    Local identifiers declared as `EXPORT` (`EXPORT` used inside a module)
    Global identifiers declared as EXPORT (EXPORT used outside any module)

Segmentation

In addition to dividing your program into code modules, you can associate groups of one or more code modules into **segments.** As your program is executed, the Macintosh Segment Loader will load all the modules in each segment at the same time, whenever any one module in the segment is called. This lets you use the same memory space for different modules as long as they are in different segments. For example, you may have a collection of modules needed only for initialization of your program. These modules could be in one segment and the rest of your program in another segment. During initialization, only the initialization segment need be loaded. After initialization, that segment can be unloaded (by a call to `UnloadSeg`) and the same memory space reused by the remainder of your program. Segments and the Segment Loader are further discussed in *Inside Macintosh.* Data initialization is discussed in the *Macintosh Programmer's Workshop Reference.*

The `SEG` directive, described in Chapter 4, lets you group a code module or a collection of code modules into a particular segment. Only code modules may be placed in segments; data modules are not affected by `SEG` directives.

Each `SEG` directive specifies a name for the succeeding segment. All code modules up to the next `SEG` directive are grouped in the specified segment, beginning at the next code-module directive.

Code modules grouped in the same segment do not have to be contiguous in the source file. Code modules belonging to different segments may be mixed in your source text as long as they are covered by the appropriate SEG directives. The SEG directive is further discussed under "Linker and Scope Controls" in Chapter 4.

❖ *Note:* Segment names are case-sensitive. Be careful to capitalize them consistently.

Machine instruction syntax

Machine instruction statements are written in four **fields**—the **label field,** the **operation field,** the **operand field,** and the **comment field.** These fields must be separated by one or more spaces (ASCII code $20) or tabs (ASCII code $09), and must be written in the order given. Total statement size is limited to 120 characters. You may continue writing a statement on the next line if you follow these rules:

- The fields must remain in their proper sequence: label, operation, operand, and comment.
- The fields must be separated by one or more spaces or tabs.
- Only the operand and comment fields may be continued. The label and operation fields must be completed in the first line of the statement, including at least one space following the operation entry.

Each continued line (after the first line) starts at the first character on that line that is not a space or tab; leading spaces or tabs on continued lines are ignored. For further information about continuing machine instruction statements, see "The Operand Field" later in this section.

The label field

The label field is the first field in a source text line. It may be empty or it may contain an identifier. The syntax rules for identifiers are given under "Identifiers" later in this chapter.

If the label field contains an identifier, it need not begin in the first character position on the line. However, if it contains an identifier that begins after the first character position (that is, if the identifier is preceded by one or more spaces or tabs), the label field must be terminated by a colon. Otherwise the label field may be terminated by either a colon, a tab, or one or more spaces. The colon, if used, is not part of the identifier.

Within code and data modules, the label field may be the sole field of a source text line, in which case it terminates with the return character that ends the line. In code modules, the Assembler always aligns such label positions to start on a word boundary, unless overridden with an `EQU` or `ALIGN 0` directive. When a source text line contains only the label field and the comment field, they must be separated by a semicolon (;) preceded by at least one space character.

Here are some examples of valid label syntax:

```
label       MOVE.W      D0,D1
 label:     MOVE.W      D0,D1
label
label                                          ;comment
```

The first line shows a label that begins in the first character position, and hence can be terminated by a tab or spaces. The second line shows a label preceded by a space; it must be terminated by a colon. The third line contains only a label. The last line contains a label and a comment, which must be separated by a semicolon preceded by at least one space or tab.

All labels that begin with an at symbol (@) are called @-labels. They can be used only inside modules. @-labels are discussed above, under "Scope of Definitions."

The Assembler allows labels for all instructions, macro calls, and directives that define data structures or values. For instructions and data-definition directives, the label is given a value equal to the location-counter value associated with the first byte of the instruction or data. For macro calls and other directives, the label's value is defined as a function of the macro call or directive.

The operation field

The operation field contains the mnemonic operation code specifying the desired machine instruction or Assembler directive. Mnemonic operation codes conform to the rules for identifiers given later in this chapter. Valid operation codes include the following:

- mnemonics for the MC68000 and MC68010 instructions described in the Motorola *M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual*
- mnemonics for the MC68020 instructions described in the Motorola *MC68020 32-Bit Microprocessor User's Manual*
- mnemonics for the MC68851 PMMU coprocessor instructions described in the Motorola *MC68851 Paged Memory Management Unit User's Manual*
- mnemonics for the MC68881 floating-point coprocessor instructions described in the Motorola *MC68881 Floating-Point Coprocessor User's Manual*
- the Assembler directives, including macro instructions, described in this book

❖ *Note:* Some mnemonics have been changed to eliminate ambiguities and conform to the Motorola assembler forms. If in doubt, check your mnemonics with those listed in Chapter 3 and Appendix G.

The operation field must be preceded by at least one space or tab. The Assembler ignores uppercase and lowercase distinctions when reading it.

Certain machine instruction mnemonics include condition codes, indicated by the symbol *cc*. A list of the condition codes the MPW Assembler accepts is included in Appendix G.

Many instructions and directives can operate on more than one data size. For these operations, the data size must be specified as part of the mnemonic; otherwise a default size is assumed. The size is specified by appending to the mnemonic a period (.) followed by one of the qualifier letters shown in Table 2-2.

Table 2-2
Data size qualifiers

Letter	Name	Data Size
B	Byte	8 bits
W	Word	16 bits
L	Long word	32 bits for data; signed offset for branch instructions
S	Short	8-bit signed offset, –128..127, for branch instructions
D	Double long word	64 bits; for certain MC68851 registers only
S	Single precision	32-bit IEEE format for binary reals: 8 exponent bits, 23 mantissa bits, 1 sign bit; MC68881 only
D	Double precision	64-bit IEEE format for binary reals: 11 exponent bits, 52 mantissa bits, 1 sign bit; MC68881 only
X	Extended	96-bit IEEE format for binary reals: 15 exponent bits, 64 mantissa bits, 1 sign bit, 16 reserved bits; MC68881 only
P	Packed BCD	96-bit packed BCD format for real strings: 3 decimal digits exponent, 17 decimal digits mantissa, 4 bits sign and range, 12 reserved bits; MC68881 only

In macro calls, the period may be followed by any sequence of characters, as long as none of them are spaces or tabs. The meaning of such a qualifier is a function of the macro definition associated with the call. See "Defining Macros" in Chapter 5 for further details.

Ordinarily, the default data size qualifier is word (W) for MC680x0 and MC68851 instructions and extended (X) for MC68881 instructions. Some instructions do not permit a data size specification, since the size is implicit in their operation.

In some cases, the Assembler accepts a **generic form** for an instruction and assembles a more appropriate form. The instruction ADD, for example, is translated into ADD, ADDA, ADDQ, or ADDI, depending on context. The generic instruction formats are listed in Appendix A. The reasons for using them fall into three overlapping categories:

- Optimization: Instructions can often be encoded into a more compact (and generally faster-executing) form that is not the same as the original instruction. An example of such an instruction is SUBA *An*,*An* in place of MOVE #0,*An*.
- Convenience: Instructions may need to be encoded based on context, such as an ADDI in place of an ADD. Also, alternate instructions may make coding easier and more readable. For example, BZ (branch if zero) would replace BEQ.
- Compatibility: Instructions may need to be translated for compatibility with the Lisa™ Workshop Assembler (TLA), the Macintosh 68000 Development System Assembler (MDS), or the Motorola Assembler. Examples here include BHS (branch on high or same) for BCC (branch on carry clear), and BLO (branch on low) for BCS (branch on carry set).

You can control whether or not the Assembler optimizes instructions by using the OPT directive, described under "Assembly Options" in Chapter 4.

The operand field

Many instructions and directives require operands as part of their specification. The operand field follows the operation field and must be separated from it by at least one space or tab. The operand field may be empty, or it may be composed of one or more subfields separated by commas.

The BLANKS Assembler directive controls where tabs or spaces may be placed within the operand field. With BLANKS OFF, they may occur only after commas separating operand subfields and between paired parentheses, brackets, or braces. With BLANKS ON, tabs and spaces may be placed anywhere in the operand field except within symbols. (Symbols are discussed under "Symbols" later in this chapter). With BLANKS ON (the preset condition), a semicolon is always required to separate the operand field from the comment field. For further information, see the discussion of BLANKS under "Assembly Options" in Chapter 4.

If you intend to continue an operand field on the next line, you must place the backslash **continuation character** (\) in the operand field before any semicolon which precedes a comment. The backslash character may not be used to continue a single symbol. This means that line continuation can occur only between symbols. Furthermore, with BLANKS OFF you must place the continuation character so that the Assembler treats it as part of the operand field—that is, immediately before or after a symbol, or among tabs or spaces that the Assembler will ignore because the operand field is not yet complete.

Here is an example of the correct way to continue an operand field:

```
EXPORT      name1,name2,name3,\
            name4,name5:DATA        ; Looks good
```

Here is an example of an incorrect operand continuation:

```
EXPORT      name1, name2, na\
            me3:DATA                ; Broken symbol
```

Comments

You can insert a comment into your source text in two ways: as a comment field or as a **comment line.** Comments are ignored by the Assembler and may contain any characters except return (ASCII $0D). Comments are intended for your use in documenting your program.

The comment field is the last field in a source text line; it must be separated from the preceding field by at least one tab or space. As mentioned above under "The Operand Field," the setting of the BLANKS directive influences whether or not comments must be preceded by semicolons. In statements where an optional opcode or operand field or subfield is omitted but a comment is desired, the comment must always be separated from the rest of the line by a semicolon preceded by one or more tabs or spaces, even with BLANKS OFF.

An entire line may be used for a comment by placing an asterisk (*) or semicolon (;) in the first character position of the line. On lines which contain only a label, the semicolon convention must be used, even with BLANKS OFF.

To continue a comment, enter a backslash continuation character (\) and go immediately to the next line.

Lines that contain no characters are ignored by the Assembler. They are treated like comment lines and can be used to separate sections of code or comments.

Here are some examples of valid comment syntax:

```
label1      MOVE.W      D0,D1           This is a comment with BLANKS OFF
label1      MOVE.W      D0,D1           ; This is a comment with BLANKS ON
label2                                  ; No opcode-semicolon required
label3      PROC                        ; Semicolon required because
                                        ; PROC has optional parameters

* This is a whole-line comment.

all blank lines are ignored

; This is also a whole-line comment.
```

```
* This is a comment that is too long to be written entirely on one line \
and therefore is continued on a second line.

* However, you can also continue a comment on a second line without using the
* continuation character, by starting the second line with another asterisk.
```

Symbols

Except for comments, all fields of an Assembler statement are composed of **symbols.** A symbol is a character or a combination of characters used to represent an identifier, a numeric constant, or a character string.

Different kinds of symbols are allowed in the different fields of assembly-language source text:

- The label field may contain only a single identifier.
- The operation field must contain a single MC68xxx instruction mnemonic, a macro call, a directive name, or an identifier defined by OPWORD.
- The operand field may contain one or more symbols and/or expressions composed of symbols of any kind.

The following sections discuss the symbols accepted by the MPW Assembler. "Expressions," later in this chapter, discusses the expressions that you can form out of symbols.

Identifiers

Names and labels are **identifiers.** The first character of an identifier must be an uppercase or lowercase letter (A..Z, a..z), an underscore (_), or an at symbol (@). The Assembler treats any label that begins with @ as an @-label.

Subsequent identifier characters can be letters, digits (0..9), underscores (_), dollar signs ($), number signs (#), percent signs (%), or at symbols (@). An identifier can be any length, but only the first 63 characters are significant. By using the CASE directive, you can specify whether uppercase and lowercase letters are to be treated as different or the same. See "Assembly Options" in Chapter 4 for more information.

Some examples of valid identifiers are shown here:

```
BYTE            ApplZone        X       A_1
Start           inverseBit      a%      A$2
NextChar        _trap           @2      A#2
Next_Char       Num#65          _       A@2
```

A special identifier symbol is used to refer to the current value of the **location counter** in a module or template. This symbol is the asterisk (*). It may appear only in the operand field. It stands for the address of the first byte of currently available storage after any required boundary alignment. Using the asterisk in the operand field of a statement is the same as placing a label in the label field of the statement and then using that label in the operand field of the same statement.

The Assembler uses the location counter referred to by the asterisk symbol to assign code and data module addresses to statements. It is the Assembler's equivalent to a computer's instruction counter. Since all modules are relocatable, all modules are assembled with their addresses relative to zero. Therefore the location counter is a zero-relative offset to the address of the start of the current module. Since it is an offset, the location counter may also be used in templates. Hence each module and template may be viewed as having its own location counter.

Numeric constants

You can express **numeric constants** in your source text in **decimal, hexadecimal, binary,** and **floating-point** form.

These are the syntax rules for expressing numeric constants:

- Decimal numbers are formed as strings of decimal digits (0..9). For example:

```
123     =     32
```

- Hexadecimal numbers are specified by a dollar sign ($) followed by a sequence of hexadecimal digits (0..9, A..F, or a..f). For example:

```
$123    $1A3C   $FFFF   $a1C2
```

- Binary numbers are specified by a percent sign (%) followed by a sequence of binary digits (0 or 1). For example:

```
%1010   %101   %1011101
```

- Floating-point numbers are specified by enclosing a decimal or hexadecimal number in quotation marks (", ASCII $22). Decimal numbers, for this purpose, may include any of the forms listed in the *Apple Numerics Manual,* Table 4-1. Hexadecimal floating-point numbers must begin with a dollar sign, following the format given above. Here are some examples:

```
"123"          "123.4E-12"          "123."          ".456"
"-0"           "-INF"               "NAN(12)"       "-Nan()"
"Nan"          "$3F800000"
```

The MPW Assembler interprets decimal, hexadecimal, and binary constants as signed 32-bit values. For example, `$FFFF` is interpreted as the value 65535, not –1. If you want –1, you must write it in decimal as `-1` or in hexadecimal as `$FFFFFFFF`. The Assembler interprets floating-point numbers as required by the MC68881 instruction that uses them—as single, double, extended, or packed BCD.

❖ *Note:* The Assembler pads incomplete hexadecimal floating-point numbers with zeros at the *right* end. For example, if you write "`$A123`" as an operand for an MC68881 instruction that requires eight-byte data, the Assembler will interpret it as the number $A1230000.

Strings

A **string** is a sequence of one or more ASCII characters (including spaces and tabs) enclosed in single quotes (', ASCII $27). Within a string, two single quotes in succession represent one single quote. When a string is written inside a macro, two ampersands (&) in succession represent a single ampersand. Some examples of strings are

```
'Hello'
'don''t'
''''                    Generates one single quote
'&&x'                   Generates one ampersand, in macros only
```

There are restrictions on how long a string can be, as well as how it is interpreted by the Assembler. These restrictions depend on its context and form, as explained below.

A **string constant** used to represent an integer value in an arithmetic expression is limited to four characters. The Assembler evaluates each character as having the value of its ASCII code. It treats such a string as a right-justified 32-bit value, padded on the left with zeros. Here are some examples:

```
SUB         #'A'-'a',D0             Constant represents the value 32
SUB         #'a'-$41,D0             Same as the previous example
MOVE.B      #'1',D0                 Put $31 into low byte of D0
MOVE.W      #'1',D0                 Put $0031 into low word of D0
MOVE.W      #'12',D0                Put $3132 into low word of D0
MOVE.L      #'1',D0                 Put $00000031 into D0
MOVE.L      #'12',D0                Put $00003132 into D0
MOVE.L      #'123',D0               Put $00313233 into D0
MOVE.L      #'1234',D0              Put $31323334 into D0
MOVE.L      #'12'+1,D0              Put $00003133 into D0
```

Strings used under any of the following conditions may be of any length:

- Strings defined as data operands to DC and DCB directives.
- Strings used in relational expressions.

- Strings used to assign values to macro variables.
- Strings used as source operands for PEA and LEA instructions. This is the only case where an arbitrary-length string may be used in a machine instruction. It represents an instance of a literal. Literals are discussed in Chapter 3 under "Special Address Formats."

Using the STRING directive, arbitrary-length string constants may be generated in any of three formats, depending on the option specified:

- **As-is string:** the string is generated as specified.
- **C string:** the string is generated with a zero-value byte following its last character. This is the the string format used by C.
- **Pascal string:** the string is preceded by a length byte. This, the default setting, is the format of strings used by Pascal and the Macintosh library routines. Pascal strings are limited to 255 characters.

For further information about the STRING directive, see "Assembly Options" in Chapter 4.

If a string variable appears as the value of a macro parameter, the Assembler interprets it as a string when it appears in a relation and as an integer when it appears in an arithmetic expression. For example, suppose the string '123' is the value of the macro parameter &i. When used in the expression &i='125' it would appear as a string. In the expression &i+10, it would yield the 32-bit value of the integer 133. This is different from the case of a string constant '123', which is treated as the value $00313233.

Declared macro string variables are always treated as strings and may be used only as strings. Macros have typed variables, as described in Chapter 6 under "SET Variables". Such variables declared as specific types may be used only in contexts appropriate to their type.

Expressions

Expressions are used either in the operand field of source text or as SET array variable subscripts (defined in Chapter 6 under "SET Array Variables"). They may be composed of a single term or a combination of terms, with each term being either an identifier, a constant, the location-counter symbol (*), or a macro function call. Integer terms are treated as 32-bit signed values, and are combined by arithmetic, logical, shift, and relational operators. Macro string terms may be combined only with relational operators.

The MPW Assembler recognizes the operators shown in Table 2-3. They are listed from lowest precedence to highest. Groupings indicate operators of the same precedence.

Table 2-3
Operators

	Symbols			Operation
Highest precedence	( )			Grouping by parentheses
	~			Ones complement
	¬	NOT		Logical not
	-			Unary negation
	*			Multiplication
	/	DIV	÷	Division
	//	MOD		Modulus division
	+			Addition
	-			Subtraction
	>>			Shift right
	<<			Shift left
	=			Equal
	<>	≠		Not equal
	<			Less than
	>			Greater than
	<=	≤		Less than or equal to
	>=	≥		Greater than or equal to
	**	AND		Logical and
Lowest precedence	++	OR	|	Logical or
	--	XOR		Logical exclusive-or

The rules for writing expressions are as follows:

- Only parentheses and the +, -, ~, ¬, and NOT operators are allowed at the start of an expression.
- Subexpressions are designated by enclosing the subexpression in parentheses.
- An expression may not contain two terms or two operators (other than parentheses) in succession.
- Parentheses may be nested to a maximum depth of 20.
- An expression may not contain more than 20 terms.
- If an expression is enclosed in parentheses, the Assembler ignores blanks within the expression regardless of the setting of the BLANKS directive.
- The multicharacter operators DIV, MOD, AND, OR, XOR, and NOT must be separated from identifiers by at least one space. Hence these operators may be combined with identifiers only if the BLANKS directive is ON (the preset mode), or if the expression containing them in enclosed in parentheses.

- Floating-point constants may be enclosed in parentheses but may not be combined with any of the other operators in Table 2-3.

Evaluation of expressions

A single symbol is a single-term expression with the value represented by the symbol. The Assembler reduces multiterm expressions to single values, following these rules:

- Each numeric term is given a 32-bit value. Overflows are ignored.
- Operations are performed from left to right, following the precedence indicated in the operator table above.
- A parenthesized subexpression is reduced to a single value. The resulting value is then used in computing the final value of the expression.
- When parenthesized subexpressions are nested, the innermost subexpression is evaluated first.
- Every expression is computed as a 32-bit signed value. The limits on the final value depend on how the expression is used.
- Division always yields an integer result; any fractional portion of the result is dropped.
- Division by 0 yields 0 as the result.
- The relational operators assign the absolute value 1 when the relation is `true`, and the absolute value 0 when the relation is `false`. The comparison is algebraic, except when two character strings are compared. See "Boolean Control Expressions" in Chapter 7 for a discussion of the rules governing string comparisons.
- The `NOT` operator is equivalent to an exclusive-or with 1—that is, $\neg e$ is equivalent to the expression *e* `XOR 1`. This lets you negate Boolean expressions containing relational operators.
- The shifting operators << and >> shift the left operand by the number of bits specified in the right operand. Zeros are shifted into vacated bit positions. Bits shifted out are lost.

Expressions used as operands for `DC` and `DCB` directives or as literal operands for `PEA` and `LEA` statements may have either string or integer values. The Assembler decides which type the expression has by checking its first symbol. If the first symbol can be interpreted as a string, the Assembler assumes the whole expression is a string. Thus the Assembler may fail to evaluate certain ambiguous expressions, such as `'a'-32`, correctly because the first symbol is a string constant. To force the Assembler to evaluate an expression arithmetically, enclose it in parentheses; for example, `('a'-32)`.

Here are some examples of valid expressions:

```
*                   A*10                (a AND b)           ~(x+10)
*+100               'a'-32              NOT(a OR b)         a>>b
Rec.Field+10        Alpha+(i>j)*10      'ab'+$8000          -10+x.y
Alpha-Beta          -64                 (a**b)++(c**d)
(a-b)/(20+(c-d))    (a-b)+(c-d)         (a AND b)OR(c AND d)
```

Absolute and relocatable expressions

When an identifier is used as a label, the Assembler assigns it a value. This value is **absolute** or **relocatable,** depending on the kind of statement or directive being labeled. Absolute values are unaffected by their code module's location in memory and have the same values at assembly time as they do at run time. Relocatable values represent addresses.

Code and data module identifiers and code or data labels are relocatable. All code modules are relocatable and data modules are relocated relative to register A5. Template identifiers and fields are absolute.

Using the location-counter symbol (*) in an Assembler statement is the same as placing a label in the label field of the statement and then using that label in the operand field of the same statement. Since using the location counter is equivalent to using the label, it may be considered either relocatable or absolute—relocatable when used in a code or data module, absolute when used in a template.

The use of absolute and relocatable values in expressions causes the expression and its resulting value to be either absolute or relocatable. The following sections describe how to create absolute and relocatable expressions.

Absolute expressions

An absolute expression may be an absolute symbol representing an absolute value, or any arithmetic combination of absolute symbols. The resulting value is an absolute value. All operators are allowed in absolute expressions, subject to the rules given above under "Evaluation of Expressions."

An absolute expression may contain relocatable values, alone or in combination with absolute terms. All terms in such an expression must already have a value; there may be no forward references. If there are relocatable terms there must be exactly one pair of them, and the relocatable terms

- may be used only in effective addresses of machine instructions and DC or DCB directive statements

- must access the same segment
- must refer both to code or both to data
- must consist of terms with opposite signs (+ and −)

The pairing of relocatable terms of opposite sign is allowed in an absolute expression because the subexpression involving the difference between the relocatable terms cancels the effect of relocation, thus producing an absolute value.

The following examples illustrate absolute expressions. In these examples *r1* and *r2* are relocatable symbols; *a1* and *a2* are absolute symbols.

```
a1
a1+100-a2
a1*a2
r1-r2
(r1-r2)+a1*10
r1-a1*10-r2
```

Relocatable expressions

A relocatable expression may contain relocatable values, alone or in combination with absolute terms, provided that it conforms to these rules:

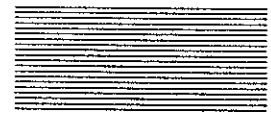
- There must be either one or three relocatable terms.
- If there are three relocatable terms, two of them must be paired, as described above under "Absolute Expressions."
- Relocatable symbols may be combined only with the + and − operators.

A relocatable expression reduces to a single relocatable value. This value is derived from the odd relocatable term, adjusted by the values of the absolute terms.

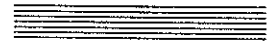
In effective addresses, the Assembler assumes that all imported code or data symbols and all forward references to undefined symbols are relocatable.

The following examples illustrate relocatable expressions. In these examples, *r1* and *r2* are relocatable symbols; *a1* and *a2* are absolute symbols; *i1* and *i2* are imported symbols.

```
r1-10
r1-(a1*10)+r2
i1
i1-10-i2
i1-10
```



Chapter 3



Address Syntax

This chapter covers the syntax rules for writing MC68000 and MC68020 addresses in Macintosh Workshop assembly-language source text.

Addressing modes

The MC68000 **effective addressing modes** are fully discussed in the Motorola *M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual.* Additional addressing modes are available for the MC68020. Both the MC68000 and MC68020 addressing modes are discussed in the *M68020 32-Bit Microprocessor User's Manual.* If you are not familiar with how the Motorola addressing modes work, you should read one of these books before trying to understand this chapter.

The symbols used in MPW assembly-language addresses are listed in Table 3-1.

Table 3-1
Address symbols

ID	Meaning
A*n*	Address register *n*, where *n* is a number in the range 0..7
D*n*	Data register *n*, where *n* is a number in the range 0..7
R*n*	Register *n*, either address or data, where *n* is a number in the range 0..7
X*n*	Index register *n*, where *n* is a number in the range 0..7. An index register may be a data register (D*n*) or an address register (A*n*), optionally followed by a period and a `W` or `L` size designation (16 or 32 bits, respectively).
**s*	Scaling factor, where *s* is an absolute expression which must produce the value 1, 2, 4, or 8. Values 2, 4, and 8 can be used only with the MC68020. The default value for *s* is 1. If you omit *s*, you must omit the asterisk also.
PC	The program counter
ae	An absolute expression
re	A relocatable expression
d	An absolute (*ae*) or relocatable (*re*) expression

bd Base displacement that is added before indirection occurs (MC68020 only). This is defined as an absolute expression for addressing mode 6 and a relocatable expression for addressing mode 73. A word or long word is generated for the *bd* as a function of its value, or for forward references, as specified by the FORWARD directive. However, you may explicitly control the generated size by using the syntax (*bd*).W or (*bd*).L.

od Outer displacement that is added after indirection occurs (MC68020 only). This is defined as an absolute expression. A word or long word is generated for the *od* as a function of its value, or for forward references, as specified by the FORWARD directive. However, you may explicitly control the generated size by using the syntax (*od*).W or (*od*).L.

Table 3-2 defines the syntax accepted for each addressing mode. Not all addressing modes are allowed for all machine instructions. The Motorola manuals cited at the beginning of this chapter tell you which addressing modes may be used with each instruction. Some of the modes have alternate syntactic forms, as shown. These alternate forms are discussed below under "Ambiguities and Optimizations."

Table 3-2
Address syntax summary

Mode	Addressing Mode	Effective Address Syntax		
0	Data register direct	D*n*		
1	Address register direct	A*n*		
2	Address register indirect	(A*n*)		
3	Postincrement register indirect	(A*n*)+		
4	Predecrement register indirect	–(A*n*)		
5	Indirect with 16-bit displacement	*d*(A*n*)		
6	Indirect with indexing plus 8-bit displacement	*d*(A*n*,*Xn*)		
* 6	Indirect with indexing plus base displacement	(*bd*,A*n*,*Xn***s*)	*bd*(A*n*,*Xn***s*)	
* 6	Indirect with preindexing	([*bd*,A*n*,*Xn***s*],*od*)		
* 6	Indirect with postindexing	([*bd*,A*n*],*Xn***s*,*od*)		
70	Absolute word (16 bits)	*ae*	(*ae*).W	
71	Absolute long (32 bits)	*ae*	(*ae*).L	
72	PC-relative with 16-bit displacement	*re*	*d*(PC)	
73	PC-relative, indexing, 8-bit displacement	*d*(D*n*)	*d*(PC,*Xn*)	
* 73	PC-relative, indexing, base displacement	(*bd*,PC,*Xn***s*)	*bd*(PC,*Xn***s*)	*bd*(*Xn***s*)
* 73	PC-relative with preindexing	([*bd*,PC,*Xn***s*],*od*)		
* 73	PC-relative with postindexing	([*bd*,PC],*Xn***s*,*od*)		

74	Immediate	*#ae*		
72	Literal (PC-relative with 16-bit displacement)	*#ae*	*#(ae).*W	*#(ae).*L

* Modes usable only with the MC68020

When writing the address forms shown in Table 3-2, you must follow these rules:

- You must write parameters in the order shown.
- The square brackets shown in Table 3-2 do not indicate optional parameters. You must write the brackets as shown.
- Expressions that specify immediate operands, literals, or absolute addresses may not contain any forward, undefined, or imported references.
- Expressions that specify immediate operands and literals, with the exception of absolute addresses, may contain SET variables and functions. You may follow the rules for absolute expressions in macro directives, explained in Chapter 5.
- Expressions that specify displacements may contain imported references but no SET variable or function references.
- Any expression involving an unpaired data reference or a forward reference to an undefined identifier is assumed to be relocatable.
- Wherever *Xn* is shown, it may be written as *Xn.*W or *Xn.*L to indicate either 16-bit or 32-bit indexing. If the size is omitted, 16-bit indexing (suffix W) is assumed.
- The scale factor **s* may be omitted. If omitted, a scale factor of 1 is assumed. If it is specified, *s* must be an absolute expression with the value 1, 2, 4, or 8. Values of 2, 4, and 8 are allowed only with the MC68020.
- The addressing form *bd*(A*n*,X*n***s*) with a scale factor *s* of 2, 4, or 8 will generate a brief MC68020 effective address format, while the form (*bd*,A*n*,*Xn***s*) will generate the full format. Similarly for the *PC*-relative forms: *bd*(PC,X*n***s*) will generate the brief format and (*bd*,PC,*Xn***s*) will generate the full format.
- The addressing form *d*(D*n*) is equivalent to *d*(PC,D*n*) and generates the effective address *d*-*(PC,D*n*). Similarly, the form *bd*(*Xn***s*) generates the effective address *bd*-*(PC,*Xn***s*).
- When two registers appear in parentheses, if the leftmost could be either A*n* or *Xn* (in other words, if no explicit scaling or size is specified), then the base register A*n* is assumed to be the leftmost and the second is assumed to be the index register *Xn*.
- Parameters may be omitted in the six MC68020 modes. If a parameter is omitted, the comma preceding it, if any, must also be omitted. Omitted registers take on suppressed register values (0). Omitted displacements or displacements with the value 0 take on null values (also 0). Omitted parameters may result in ambiguous addressing modes, as discussed below under "Ambiguities and Optimizations." However, you can resolve these ambiguities by using zero-suppressed registers.

- Register mnemonics ZPC, ZA0-ZA7, and ZD0-ZD7 specify zero-suppressed registers—registers whose values are treated as 0 during effective address calculations. These symbols may be used to specify any allowable register in the six MC68020 addressing modes. Using such mnemonics also explicitly forces one of the extended addressing modes, if omitting the specified registers would cause the Assembler to substitute a simpler mode. For the rules covering such substitutions see "Ambiguities and Optimizations," below.
- Equates (EQU and SET directive statements) to absolute values (such as constants and registers) must be written before the equated symbols are used in the source text. When the Assembler encounters a symbol in an effective address, the Assembler first looks for the symbol in the code module's local symbol table. If it does not find the symbol there, it searches for it in the global symbol table.
- Since the Assembler computes all expressions as 32-bit values, you must be careful when writing displacement in hexadecimal. The effective address $FFF6(A5), for example, is equivalent to 65526(A5), not -10(A5), and is therefore illegal. To specify an offset of -10, you must write either -10(A5), or -$A(A5), or $FFFFFFF6(A5).
- Mode 70 and 71 addressing modes (absolute word and long) generate 16 or 32 bits depending on the value of the absolute expression. If you want to force a specific size, use the alternate notation. For example, by writing ($1234).L you can force the value of $1234 to 32 bits.
- With the Macintosh, all data references are based on register A5. This means that mode 5 or 6 addresses should specify A5 as the base register. With data record fields or imported templates, such a specification takes the form *record.field*(A5) or *record.field*(A5,*Xn*), where *record* is a data module record identifier and *field* is a field identifier within the record. If you specify a data field reference without an explicit base register, the Assembler will assume register A5 and will change the addressing mode to mode 5. It will do this even though your address syntax may look like a PC-relative mode.

Ambiguities and optimizations

Under certain conditions, the Macintosh Assembler will transform the address syntax you write in your source text to a simpler form. It does this to remove ambiguities, reduce object code, and improve execution speed. These automatic transformations are summarized in Table 3-3.

Table 3-3
Effective address transformations

Original form	Condition for optimization	Optimized form
(*bd*,A*n*,*Xn*s*)	Size of *bd* ≤ 8 bits	*bd*(A*n*,*Xn*s*)
(A*n*,*Xn*s*)	Omitted *bd* (*bd* = 0)	0(A*n*,*Xn*s*)
(*bd*,PC,*Xn*s*)	Size of *bd* ≤ 8 bits	*bd*(PC,*Xn*s*)
(*bd*,A*n*)	Size of *bd* ≤ 16 bits	*bd*(A*n*)
(*bd*,PC)	Size of *bd* ≤ 16 bits	*bd*(PC)
d(A*n*)	*d* = 0	(A*n*)

Here are the rules by which the Assembler performs automatic address transformation during assembly:

- The syntax for the mode 6 MC68020 extended addressing form, (*bd*,A*n*,*Xn*s*), is identical to the mode 2 addressing mode, (A*n*), if the displacement and index are omitted. If just the index is omitted, and the displacement is 16 bits or less, then the mode 6 form, (*bd*,A*n*), is identical to mode 5, *d*(A*n*). Even when the index is specified, if the displacement is eight bits or less then the mode 6 form, (*bd*,A*n*,*Xn*), is identical to the form *d*(A*n*,*Xn*). The Assembler resolves these ambiguities by selecting the more efficient forms.
- A similar situation exists for the mode 73 form, (*bd*,PC,*Xn*s*). It is identical to mode 72, *d*(PC), when the index is omitted and the displacement is 16 bits or less, and to mode 73, *d*(PC,*Xn*), when the displacement is eight bits or less. As in the mode 6 cases, the Assembler chooses the more efficient mode 72 form.
- The Assembler optimizes each address before it checks to see if it is an MC68020 address. If an MC68020 address is optimized to an MC68000 form, the Assembler will accept it even when the target microprocessor is not the MC68020. Hence you can use MC68020 forms to write MC68000 addresses whenever they meet the criteria for optimization.

Normally, the Assembler tries to use the shorter and more efficient form when interpreting the foregoing addressing modes. If you want to preserve the extended address form, use the `OPT` directive described under "Assembly Options" in Chapter 4 to suppress transformation. In this case, remember that preindexing is more efficient than postindexing, and using an index register is more efficient than using a base register for indirect modes.

If you want to override automatic mode transformations with an individual instruction and explicitly force a specific mode, use zero-suppressed registers (ZPC, ZA0..ZA7, ZD0..ZD7). But remember that when the program counter is zero-suppressed (ZPC), its displacement is assumed to be absolute and hence is not offset from the current location-counter (PC) value.

Forward-reference addressing

The size of the displacement values in the various modes of effective addresses can be 8, 16, or 32 bits depending on the value and the mode. When you use imported or forward- referenced identifiers in addresses, and there is no other way to determine their size, the Assembler assumes default sizes for the various displacements. All such default actions may be overridden with the BRANCH and FORWARD assembly-control directives.

The B*cc*, BSR, BRA,FB*cc*, and PB*cc* instructions contain 8-bit, 16-bit, and 32-bit PC-relative displacements without any explicit mode indication in their syntax. The 32-bit displacement is available only in the MC68020, and the FB*cc* and PB*cc* instructions are limited to 16-bit and 32-bit forms. The Assembler assumes a 16-bit forward-referenced offset, unless a period and suffix S or L is written after the mnemonic. The BRANCH directive allows you to change the default size.

The FORWARD directive controls the default size for all other forward-referenced displacement encodings—offsets, base displacements, and outer displacements. The default size is 16 bits; you can change this to 32 bits with the MC68020 only.

Registers

The addressing modes defined in Table 3-2 use the standard MC68000 address registers (A0 through A7), data registers (D0 through D7), and program counter (PC). Besides these, the other processors and coprocessors supported by the Macintosh Workshop Assembler contain additional registers, which may be named in some instructions and not in others. Table 3-4 lists all the registers recognized by the Assembler, including those already discussed. Refer to the appropriate Motorola manuals, listed in the preface of this manual, for their exact formats and uses.

Table 3-4
Registers

Designation	Usage
D0..D7	Data registers
ZD0..ZD7	Zero-suppressed data registers
A0..A7	Address registers
ZA0..ZA7	Zero-suppressed address registers
A7, SP	The current stack pointer
SR	Status register
CCR	Condition code register

USP	User stack pointer
MSP, SSP	Master stack pointer
PC	Program counter
ZPC	Zero-suppressed program counter
MC68010 and MC6820 only	
SFC	Source function code register
MC6820 only	
ISP	Interrupt stack pointer
DFC	Destination function code register
VBR	Vector base register
CACR	Cache control register
CAAR	Cache address register
MC68851 only	
CRP	CPU root pointer register
SRP	Supervisor root pointer register
DRP	DMA root pointer register
PCSR	PMMU cache status register
TC	Translation control register
AC	Access control register
CAL	Current access level register
VAL	Validate access level register
SCC	Stack change control register
PSR	PMMU status register
BAD0..BAD7	Breakpoint acknowledge data registers
BAC0..BAC7	Breakpoint acknowledge control registers
MC68881 only	
FP0..FP7	Floating-point data registers
FPCR	Floating-point control register
FPSR	Floating-point status register
FPIAR	Floating-point instruction address register

Any of the register names listed in Table 3-4 may be equated to other identifiers by using the EQU and SET directives. If you do this, make sure that you write the equates before you use the new symbols. When the Assembler encounters a symbol in an effective address position that may be a register, it first looks for the symbol in the code module's local symbol table. If it doesn't find it there, it searches the global symbol table.

Special address formats

Most MC68xxx instructions contain two effective addresses separated by a comma. The first address is called the **source** and the second the **destination.** The instructions generally cause an operation to be performed on the source, possibly in combination with the destination, and place a result in the destination. However, there are some exceptions to this format. This section gives the syntax rules for such exceptions.

In the following sections, *ea* represents any effective address format that may legally be used with the instruction being discussed.

MC68xxx Instructions

MOVEM: Multiple moves

```
MOVEM.size      rlist,ea
MOVEM.size      ea,rlist
size ::= W | L
```

The MOVEM instruction takes a **register list,** *rlist,* as either a source or destination. The register list syntax is as follows:

- *Rm–Rn* designates registers *Rm* through *Rn* (where $m \leq n$, and *Rm* and *Rn* are both A registers or both D registers).
- *Ri/Rj/Rk...* designates registers *Ri, Rj, Rk...* where each term is an A register, a D register, or a range *Rm–Rn.*

Here are two examples:

D0-D1/A3	means	D0, D1, and A3
D2-D4/A1-A2/D7	means	D2, D3, D4, A1, A2, and D7

MC68020 instructions

For details of the syntax of these instructions, see the Motorola *MC68020 32-Bit Microprocessor User's Manual.*

MULS and MULU: Signed and unsigned multiplication

`MULS.L`	*ea*,D*l*	32 x 32 --> 32
`MULS.L`	*ea*,D*h*:D*l*	32 x 32 --> 64
`MULU.L`	*ea*,D*l*	32 x 32 --> 32
`MULU.L`	*ea*,D*h*:D*l*	32 x 32 --> 64

In these syntax diagrams, D*l* designates the low-order register (*l*=0..7) and D*h* the high-order register (*h*=0..7). These instructions support 32-bit multipliers and a 32-bit or 64-bit product, as shown by the comments in the far right column.

DIVS and DIVU: Signed and unsigned division

`DIVS.L`	*ea*,D*q*	32/32 --> 32q
`DIVS.L`	*ea*,D*r*:D*q*	64/32 --> 32r:32q
`DIVU.L`	*ea*,D*q*	32/32 --> 32q
`DIVU.L`	*ea*,D*r*:D*q*	64/32 --> 32r:32q

In these syntax diagrams, D*q* designates the quotient register (*q*=0..7) and D*r* the remainder register (*r*=0..7). These instructions support a 64-bit dividend, a 32-bit quotient, and a 32-bit remainder, as shown by the comments in the far right column.

TDIVS and TDIVU: Truncated signed and unsigned division

`TDIVS.L`	*ea*,D*q*	32/32 --> 32q
`TDIVS.L`	*ea*,D*r*:D*q*	32/32 --> 32r:32q
`TDIVU.L`	*ea*,D*q*	32/32 --> 32q
`TDIVU.L`	*ea*,D*r*:D*q*	32/32 --> 32r:32q

In these syntax diagrams, D*q* designates the quotient register (*q*=0..7) and D*r* the remainder register (*r*=0..7). These instructions divide two 32-bit values and return either a quotient and a remainder or just a quotient, as shown by the comments in the far right column.

❖ *Note:* The current edition of the Motorola MC68020 manual uses the mnemonic DIVSL.L to refer to these instructions.

PACK and UNPK: Packing and unpacking

PACK	–(A*x*),–(A*y*),#*adjustment*
PACK	D*x*,D*y*,#*adjustment*
UNPK	–(A*x*),–(A*y*),#*adjustment*
UNPK	D*x*,D*y*,#*adjustment*

These instructions pack and unpack BCD digit formats between the source (*x*) and destination (*y*) registers. The *adjustment* is a 16-bit absolute expression added to the source value to allow character translation. This expression follows the same rules as those for immediate operands.

CAS and CAS2: Comparing and swapping

CAS.*size*	D*c*,D*u*,*ea*
CAS2.*size*	D*c1*:D*c2*,D*u1*:D*u2*,(*Rn1*):(*Rn2*)
size ::= B | W | L	

These instructions are most easily explained as if they were a sequence of pseudo-Pascal statements:

```
CAS:    IF Dc=ea^
        THEN                                {We have a match,}
          ea^ := Du                         {Copy Du to ea^,}
        ELSE                                {no match}
          Dc := ea^;                        {Copy ea^ to Dc:}

CAS2:   IF (Dc1 = Rn1^) AND (Dc2 = Rn2^)
        THEN                                {We have a match,}
          BEGIN                             {set destination}
            Rn1^ := Du1;                    {Copy Du1 to Rn1^,}
            Rn2^ := Du2                     {Copy Du2 to Rn2^,}
          END
        ELSE                                {no match}
          BEGIN
            Dc1 := Rn1^;                    {Copy Rn1^ to Dc1,}
            Dc2 := Rn2^                     {Copy Rn2^ to Dc2,}
          END;
```

Both instructions operate the same way, except that CAS2 operates on two sets of registers simultaneously while CAS operates on only one set of registers.

Bit field instructions

BFCHG	*ea*{*offset*:*width*}
BFCLR	*ea*{*offset*:*width*}
BFEXTS	*ea*{*offset*:*width*},D*n*
BFEXTU	*ea*{*offset*:*width*},D*n*
BFFFO	*ea*{*offset*:*width*},D*n*
BFINS	D*n*,*ea*{*offset*:*width*}
BFSET	*ea*{*offset*:*width*}
BFTST	*ea*{*offset*:*width*}

These instructions operate on a string of consecutive bits in a bit array. In the syntax given, the braces and colon must be included as shown. A comma may be used in place of the colon. The *offset* and *width* parameters must be either data registers or absolute expressions. If they are expressions, they must follow the same rules as those for immediate operands.

Tcc and TPcc: Trap on condition

T*cc*	
TP*cc*.*size*	#*ae*
size ::= W | L	

These instructions are the same as described in the MC68020 manual except that the single mnemonic, TRAP*cc*, has been changed to two mnemonics, T*cc* and TP*cc*. T*cc* is used for the parameterless form, while TP*cc* is used when an immediate data operand is specified.

MC68881 instructions

For details of the syntax of these instructions, see the Motorola *MC68881 Floating-Point Coprocessor User's Manual.*

FMOVEM with explicit register lists

FMOVEM.*size*	*fp-rlist*,*ea*
FMOVEM.*size*	*ea*,*fp-rlist*
size ::= L | X	

The FMOVEM instruction takes a floating-point register list, *fp-rlist*, as either a source or a destination. Here are the rules of register list syntax:

- FP*m*–FP*n* designates registers FP*m* through FP*n* where $m \leq n$.
- FP*i*/FP*j*/FP*k*... designates registers FP*i*, FP*j*, FP*k*.... Each term is either a single register FP*n* or a range FP*m*–FP*n*.
- FPCR/FPSR/FPIAR designates the floating-point control registers FPCR, FPSR, and FPIAR, in any order. You cannot combine these registers with other registers in a register list.

Here are two examples:

FP0–FP3/FP7	means	FP0, FP1, FP2, FP3, and FP7
FPCR/FPSR	means	control registers FPCR and FPSR

FMOVE with packed BCD data

```
FMOVE.P     FPn,ea
FMOVE.P     FPn,ea{#k}
FMOVE.P     FPn,ea{Dn}
```

When writing this instruction you must include the braces as shown. The numerical expression inside the braces is the k-factor, which tells the MC68881 coprocessor in what format to construct the resulting decimal string. It may be expressed either dynamically (as the value in register D*n*), as an absolute expression preceded by the pound sign (#), or by default. For an explanation of k-factors, see the discussion of FMOVE in the Motorola *MC68881 Floating-Point Coprocessor User's Manual.*

FSINCOS: Simultaneous sine and cosine

```
FSINCOS.size    ea,FPc:FPs
FSINCOS.X       FPm,FPc:FPs
size ::= B | W | L | S | D | X | P
```

FP*c* is the floating-point register holding the cosine result. FP*s* is the floating-point register holding the sine result. FP*m* is the floating-point register holding the source value.

FTcc and FTPcc: Floating-point trap on condition

```
FTcc
FTPcc.size      #ae
size ::= W | L
```

These instructions are the same as FTRAP*cc*, described in the Motorola MC68881 manual, except that the two mnemonics FT*cc* and FTP*cc* have been substituted for FTRAP*cc*. You write FT*cc* for the form without parameters and FTP*cc* for the form with an operand.

FTEST: Test operand and set floating-point condition codes

```
FTEST.size      ea
FTEST.X         FPn
size ::= B | W | L | S | D | X | P
```

The FTEST instruction is the same as FTST, described in the Motorola MC68881 manual. The mnemonic was changed to FTEST to avoid ambiguity with the FT*cc* instruction using a signaling true (ST) conditional predicate.

MC68851 instructions

If your source text contains code for the MC68851 Paged Memory Management Unit coprocessor, you may use special operand formats with the instructions listed in Table 3-5. For details of the syntax of these instructions, see the Motorola *MC68851 Paged Memory Management Unit User's Manual.*

Table 3-5
Special MC68851 operand formats

Opcode	Operand format	Sizes	Notes
PB*cc.size*	*label*	*W* | *L*	
PDB*cc.size*	Dn,*label*	W	
PFLUSH	*fc,#ae*[,*ea*]		1
PFLUSHA			
PFLUSHS	*fc,#ae*[,*ea*]		1
PFLUSHR	*ea*	D	4
PLOADR	*fc,ea*		1

PLOADW	*fc,ea*		1
PMOVE	*PMMU-reg,ea*	B | W | L | D	2
PMOVE	*ea,PMMU-reg*	B | W | L | D	2,4
PRESTORE	*ea*		
PSAVE	*ea*		
PS*cc*	*ea*	B	
PTESTR	*fc,ea,#ae*[,A*n*]		1
PTESTW	*fc,ea,#ae*[,A*n*]		1
PT*cc*			3
PTP*cc*	*#ae*	W | L	3
PVALID	VAL,*ea*	L	2
PVALID	A*n,ea*	L	

1. The function code is defined as follows:

   fc ::= *#ae* (specified as four bits in the command word)
   D*n* (contained in the lower four bits of D*n*)
   SFC (contained in the processor's source function register)
   DFC (contained in the processor's destination function code register)

2. The MC68851 registers are listed in Table 3-4.

3. The Assembler recognizes the instruction mnemonics PT*cc* and PTP*cc* in place of the Motorola mnemonic PTRAP*cc*. Use PT*cc* for the form without parameters and PTP*cc* for the form with an immediate data operand.

4. The root pointer registers in the MC68851 contain double long words, 64 bits long. The FLUSHR and PMOVE instructions, which reference these registers, accept immediate effective addresses; hence for *ea* you can use *#ae* (mode 74, Table 3-2). If you do this, however, the Assembler will convert the 32-bit effective address to a 64-bit value by filling it on the left with 32 zero bits. It will also issue a warning, because it does not support 64-bit values. You can avoid this limitation by defining a constant with two DC.L directives, then referencing them in your FLUSHR or PMOVE instruction.

Literals

In the MC68xxx instruction set, the effective addressing modes for the PEA and LEA instructions do not permit immediate data. However, both these instructions are used quite often for passing parameters to subroutines, particularly for passing string addresses to Macintosh ROM routines. Hence the MPW Assembler allows you to specify immediate data to PEA and LEA instructions. This immediate data may be an absolute expression (with no forward, imported, or undefined references) or a string. It is called a **literal.** The syntax for writing PEA and LEA instructions with literals is as follows:

```
        PEA     #data               immediate data for PEA
        LEA     #data,An            immediate data for LEA
```

This is functionally equivalent to the following:

```
        PEA     L1
        LEA     L1,An
        ...
        ALIGN   2       ; Literal data always on word boundary
L1      DC.size data
```

In the last line above, the *size* qualifier is W or L, corresponding to the *data* size or to the explicit specification of the literal.

The Assembler creates a PC-relative mode 72 address when processing PEA and LEA instructions. This is legal for these instructions; the immediate addressing mode 74 is not.

All literals encountered during the assembling of a code module are accumulated in an area called the **literal pool.** Multiple references to the same literal will address only one instance of the literal in the literal pool; duplicate literals are not generated. The literal pool is attached to the end of the code module as part of the code.

When a string literal is generated, it may be any one of the three formats for character strings—an as-is string, a C string, or a Pascal string. The Assembler determines which format to use by the current setting of the STRING directive at the time the literal is placed in the literal pool by the PEA or LEA instruction. A STRING directive setting at the end of the module has no effect on the format of strings in the literal pool.

If an absolute expression is used to generate a literal, the size of the literal depends on its value, as follows:

- Values between –32767 and +65535 are created as word-size literals.
- All other integer values are created as long-word literals.

- Floating-point values are created as extended (12-byte) literals. You may use such values only if the `MC68881` directive is in force.

Because both strings and absolute expressions may be used as literals, the Assembler may interpret an absolute literal incorrectly if its first symbol is a string constant. To force the Assembler to treat a literal as absolute, enclose it in parentheses.

The Assembler lets you override the implicit sizing of numeric literals by explicitly specifying their size. This is done by extending the literal syntax, as follows:

`PEA`	`#(`*ae*`).W`	immediate word data for `PEA`
`PEA`	`#(`*ae*`).L`	immediate long-word data for `PEA`
`PEA`	`#(`*ae*`).S`	immediate single-precision data for `PEA*`
`PEA`	`#(`*ae*`).D`	immediate double-precision data for `PEA*`
`PEA`	`#(`*ae*`).X`	immediate extended data for `PEA*`
`PEA`	`#(`*ae*`).P`	immediate packed BCD data for `PEA*`
`LEA`	`#(`*ae*`).W,A`*n*	immediate word data for `LEA`
`LEA`	`#(`*ae*`).L,A`*n*	immediate long-word data for `LEA`
`LEA`	`#(`*ae*`).S,A`*n*	immediate single-precision data for `LEA*`
`LEA`	`#(`*ae*`).D,A`*n*	immediate double-precision data for `LEA*`
`LEA`	`#(`*ae*`).X,A`*n*	immediate extended data for `LEA*`
`LEA`	`#(`*ae*`).P,A`*n*	immediate packed BCD data for `LEA*`

* Available only if the `MC68881` directive is in force

Enclosing the literal in parentheses and following it with a size qualifier (such as `W`) establishes its size. The value of the literal must always lie within the range specified. Size qualifiers are described in Table 2-2.

Chapter 4

Assembler Directives

Directives are instructions to the Macintosh Workshop Assembler to perform specific operations during assembly.

A number of Macintosh Workshop Assembler directives (RECORD, PROC, EXPORT, SEG, and so on) were mentioned in Chapter 2. This chapter covers them and others in detail. The discussion is organized into these groups:

Code and data module definitions

PROC	Begin a procedure code module
ENDPROC	End a procedure code module
FUNC	Begin a function code module
ENDFUNC	End a function code module
MAIN	Begin a main program code module
ENDMAIN	End a main program code module
RECORD	Begin a data module
ENDR	End a data module
CODE	Switch assembly from data to code
DATA	Switch assembly from code to data
END	End the whole assembly

Symbol definitions

EQU	Assign a permanent value to a symbol
SET	Assign a temporary value to a symbol
REG	Assign an identifier to a processor register list
FREG	Assign an identifier to an MC68881 register list
OPWORD	Assign an identifier to an opcode

Data definitions

DC	Place constants in code or data
DCB	Place a block of constants in code or data
DS	Define a storage area

Template definitions

RECORD	Begin a record template definition
ENDR	End a record template definition
WITH	Begin default record identifier qualification
ENDWITH	End default record identifier qualification

Linker and scope controls

EXPORT	Make entry points accessible in other assemblies
ENTRY	Make local entry-points global
IMPORT	Identify entry points declared externally
CODEREFS	Control the linking of code-to-code references
DATAREFS	Control the linking of data-to-code and data-to-data references
SEG	Specify the current code segment
COMMENT	Place a comment in the object file

Assembly options

MACHINE	Identify the target microprocessor model
MC68881	Control the assembly of floating-point coprocessor instructions
MC68851	Control the assembly of PMMU coprocessor instructions
STRING	Control the encoding of string constants
BRANCH	Control the encoding of branch instructions
FORWARD	Control the encoding of forward references
OPT	Control the level of code optimization
CASE	Control the treatment of lowercase letters in identifiers
BLANKS	Control the treatment of spaces and tabs in the operand field

Location-counter controls

ALIGN	Advance the location counter to the next multiple of a value
ORG	Set the value of the location counter

File Controls

INCLUDE	Insert source text from another file
DUMP	Write the current global symbol table to a file
LOAD	Read a file into the current global symbol table
ERRLOG	Create an error-listing file

Listing Controls

PAGESIZE	Specify the listing page size
TITLE	Define a title for the listing header
PRINT	Control miscellaneous listing options
EJECT	Start a new page in the listing
SPACE	Insert blank lines in the listing

In addition to the directives listed above, the Macintosh Workshop Assembler supports directives for macro definition and expansion, macro variables, and conditional assembly. These are described in Chapters 5, 6, and 7.

Directive formats

Macintosh directives follow the general format for Assembler statements. You write them in four fields, separated by spaces or tabs, as described in Chapter 2 under "Machine Instruction Syntax":

label field	**operation field**	**operand field**	**comment field**
[*identifier*]	*directive name*	[*directive parameters*]	[*comments*]

The identifier in the label field may be required by the directive or may be an optional macro label. If you include a macro label, you can reference the directive from macro statements as described under "GOTO, IF...GOTO and Macro Labels: Branching" in Chapter 7.

If the directive does not require a label or allow an optional macro label, you cannot include a label with it.

The directive name specifies which directive the Assembler executes. It is always required. The Assembler makes no distinction between uppercase and lowercase letters in directive names.

The operand field contains the directive's parameters, if any. Such parameters may be either required or optional, depending on the directive.

You can include a comment with a directive, writing it after all the directive's required parameters (if any). With directives that have no parameters or have only optional parameters, you can still include comments even if you don't specify any parameters. Use the standard convention of placing a semicolon in the operand field, following it with the comment field. Directive comments are ignored by the Assembler.

❖ *Note:* In the remainder of this chapter and in Chapters 6 and 7, directive syntax is usually defined by means of syntax diagrams. You can find the rules for interpreting these diagrams under "Syntax Diagram Conventions" in the Preface.

Code and data module definitions

The section "Source Text Structure" in Chapter 2 describes how Macintosh object files are built from code and data modules. The directives described in this section delimit the code and data parts of your source text and tell the Assembler how to apply the statements in each part to specific modules. They are the following:

PROC	Begin a procedure code module
ENDPROC	End a procedure code module
FUNC	Begin a function code module

ENDFUNC	End a function code module
MAIN	Begin a main program code module
ENDMAIN	End a main program code module
RECORD	Begin a data module
ENDR	End a data module
CODE	Switch assembly from data to code
DATA	Switch assembly from code to data
END	End the whole assembly

PROC and ENDPROC: Define a Procedure Code Module

```
[name]          PROC        [{ENTRY|EXPORT}]
                statements
[macro-label]   ENDP[ROC]
```

A PROC directive in your source text marks the beginning of a code module. The code module extends from the PROC directive until the next ENDPROC, or until the start of the next code module (PROC, FUNC, or MAIN), the next data module (RECORD), or the end of the assembly (END). You can write ENDPROC as ENDP. Code modules are the only places where you can write machine instruction statements.

If you write a name in the label field of a PROC directive, it becomes the identifier of the code module that begins there. The identifier is global to the assembly file. If you do not provide an identifier, you must define entry points inside the module by using ENTRY and/or EXPORT directives. ENTRY and EXPORT are described later in this chapter.

You can declare the code module itself as EXPORT in two ways: by specifying the module identifier in an EXPORT directive before you define the module or by writing EXPORT as an operand in the PROC directive itself. In either case the PROC directive that begins an EXPORT code module must include an identifier in its label field.

If you do not declare a code module as EXPORT, the Assembler declares it as ENTRY by default. For clarity of documentation, you may explicitly declare it as ENTRY in the PROC directive or specify its identifier in an ENTRY directive before defining the module. The latter technique is useful if you need to make a forward reference to the module.

You can declare the identifier of a procedure code module as MAIN by using a previous ENTRY or EXPORT directive. This has the same effect as declaring it with the MAIN directive described later in this chapter.

Labels defined inside a code module are local to that module. The only way to make these labels accessible to other modules is to declare them as EXPORT or ENTRY inside the module.

FUNC and ENDFUNC: Define a Function Code Module

```
[name]          FUNC        [{ENTRY | EXPORT}]
                statements
[macro-label]   ENDF[UNC]
```

FUNC and ENDFUNC act exactly the same as PROC and ENDPROC. They are included for documentation purposes only, so that you can indicate that the code module is a function rather than a procedure. You can write ENDFUNC as ENDF.

MAIN and ENDMAIN: Define a main program code module

```
[name]          MAIN        [{ENTRY | EXPORT}]
                statements
[macro-label]   ENDMAIN
```

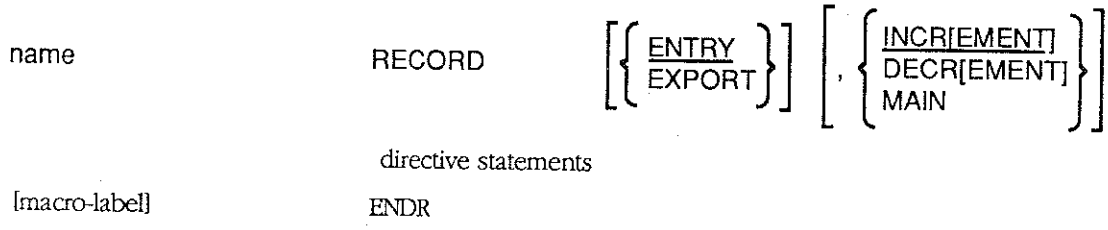
MAIN and ENDMAIN act exactly like PROC and ENDPROC, except that they declare the code module that they define as the main program. The first executable statement of that code module becomes the execution entry point for the whole program.

To declare a code module statement other than the first executable statement as a main entry point, use a previous ENTRY or EXPORT directive. ENTRY and EXPORT are described later in this chapter.

An assembly, including all its linked parts, may have only one main program module or main entry point.

❖ *Note:* If your program contains one or more data modules containing DC or DCB directives, you must link it with the library file `Runtime.o`, which contains the data initialization routine `_DataInit`. If your main code module is written in assembly language, its first executable statement must be a call (JSR) to the entry point `_DataInit`. This entry point must also be declared as `IMPORT`. After returning from `_DataInit`, your program may unload the segment `%A5Init` that contains it, by calling the Macintosh routine `UnloadSeg`. If your main program is written in C or Pascal, no explicit call to `_DataInit` is required, because the run-time libraries for C and Pascal automatically take care of data initialization.

RECORD and ENDR: Define a data module



RECORD and ENDR let you delimit and name a data module. The data module extends from the RECORD directive until the next ENDR or until the start of the next code module (PROC, FUNC, MAIN), the next data module or template (RECORD), or the end of the assembly (END). RECORD and ENDR act like PROC and ENDPROC, but define a data module instead of a code module.

❖ *Note:* RECORD and ENDR are also used to define templates. This usage is described below under "Template Definitions."

Data modules may contain only directives. Some of these directives—ORG, ALIGN, DC, DCB, and DS—define data fields. Others, such as symbol definitions, define data within the fields.

Every data module must have an identifier. This is because the identifier is used to qualify the module's field identifiers when they are accessed from code modules. Unlike labels in code modules, the field labels in data modules may be accessed by all code modules that follow them in the source text file. You can also make them accessible to other files by including EXPORT directives inside the module. Conversely, you must define a data module before accessing any of its fields in the same file.

You can access a field in a data module by an identifier of the form *mod.field*, where *mod* is the identifier of the data module and *field* is a label inside the module.

Data modules may be declared as EXPORT or ENTRY just like code modules. You can either specify the module identifier in an EXPORT or ENTRY directive before defining the module, or include EXPORT or ENTRY as an operand in the RECORD directive itself. If you do not specify one or the other, the Assembler will declare the data module as ENTRY.

The Macintosh Workshop Linker collects all data modules so that they may be loaded as a group by the Segment Loader. It loads them just below the application parameters, pointed to by A5. Thus all the data modules are accessed as A5-relative with negative offsets determined by the Linker. The implied base register for data references is A5; it need not be specified in machine instructions unless indexing is used.

As with code modules, data modules have their own location counter; it points to the next available data location in the data module. In any data module, the value of this counter may range from –32768 to +32767.

INCREMENT and DECREMENT

In code modules, each machine instruction is executed immediately after the one preceding it. Thus the location counter for a code module is incremented for each instruction. Data module location counters act similarly, except that you can choose whether they increment or decrement.

INCREMENT is the default action for any data module location counter. If you specify it in a RECORD directive or omit the parameter altogether, the Assembler will increment the resulting data module's location counter by the size of each piece of data after that data is allocated. The location counter will therefore always point to the lowest address of the next piece of data.

If you specify DECREMENT in a RECORD directive, the Assembler will allocate data in the data module in the reverse direction. This corresponds to the allocation algorithm of Pascal. The location counter is first decremented by the size of each piece of data, before it is allocated; hence each piece of data starts at an address lower than the one before it.

When the Assembler defines a DECREMENT data module, it locates the module's identifier at an entry point outside the module and at a higher address. Thus the actual module is *anonymous*; you cannot access it directly. Further, since the Assembler gives the module's identifier an offset that is equal to the size of the module, the identifier remains undefined until the completion of the module's definition. This means it cannot be accessed inside the module by expressions that require all identifiers to be previously defined, such as equates.

In terms of their actual structure in the object file, all code and data modules containing *n* bytes are considered to have their bytes numbered positively from 0 to *n*–1. With code modules and incrementing data modules, their location-counter offsets correspond directly to their object file numbering. With decrementing data modules, however, their location-counter offsets and object file numbering are complementary. Byte 0 in the object file corresponds to the last byte of the last piece of data in the module.

To illustrate this, suppose we defined a data module consisting of three long words:

Location counter				**Object file bytes**
	Data	RECORD	,DECREMENT	**12**
-4	a	DS.L	1	**8**
-8	b	DS.L	1	**4**
-12	c	DS.L	1	**0**
		ENDR		

The numbers on the left are the generated offsets as determined by the location counter. The numbers on the right are the module offsets in the object file. Since the Assembler generates references to the module as offsets from the module identifier, the Assembler's negative offsets will work only if we define the identifier as byte *n* of the module (not as byte 0), where *n* is the size of the module—in the above example, 12. In this way the identifier specifies an entry point in an anonymous module.

You can write INCREMENT as INCR and DECREMENT as DECR.

MAIN

The parameter value MAIN in a RECORD directive generates a special form of decrementing data module, called the **main data module.** When it collects all the data modules in your source text together, the Linker normally adjusts the A5 offsets in all code statements that access data. This means that the Linker must retrieve all referenced data locations from the object file. By declaring one data module as MAIN, you can shorten this process.

A program can have only one main data module. The Segment Loader loads it first, immediately below A5. Because the position of the main data module is unique, the Assembler can adjust the code statement offsets that access it without relying on the Linker. The Linker, in turn, does not retrieve the locations of data in the main data module and no Linker records are generated for it in the object file. As a result, the unlinked object file is smaller and the Linker runs faster.

Because the main data module is loaded below A5 and its offsets are generated by the Assembler, the generated offsets are negative. Therefore the main data module is always a decrementing data module.

CODE and DATA: Switch between code and data

```
[macro-label]        CODE

[macro-label]        DATA    [ { INCR[EMENT] } ]
                             [ { DECR[EMENT] } ]
                             [ { MAIN        } ]
```

You can define an associated data module during the definition of a code module, without ending the current code module, by using CODE and DATA. (This technique is illustrated in Chapter 2 under "Source Text Structure.")

CODE and DATA may be used only inside a code module—a module defined by PROC, FUNC, or MAIN. DATA switches the Assembler to defining a data module; CODE switches it back to defining the original code module. The final result is one contiguous code module and one contiguous data module, regardless of how many times you use CODE and DATA. Remember that your source text may contain only directive statements when DATA is in force; it may contain both machine instruction statements and directive statements when CODE is in force.

The DATA directive can generate either an incrementing, decrementing, or main data module, depending on the value of its parameter. With no parameter, it generates an incrementing data module. A full explanation of these options is given under "RECORD and ENDR" above. The option you select the first time you use DATA in a given code module governs all data generation within that module; the Assembler will ignore subsequent DATA parameters until the code module ends. You can generate different kinds of data modules from different code modules, however. You can use MAIN, with either RECORD or DATA, only once in a program.

You can write INCREMENT as INCR and DECREMENT as DECR.

END: End the assembly

```
[macro-label]        END
```

The END directive marks the end of your assembly. The Assembler ignores any source text after END.

END is a required directive. The Assembler generates a warning (not an error) if it is omitted. You must not place END in a file called by an INCLUDE directive, unless you intentionally want to terminate your assembly from the included file.

Symbol definitions

The directives described in this section let you assign values to individual identifiers. They let you name certain objects—numeric constants, individual registers, register lists, and opcodes—so that you can use the identifiers instead of the original objects in your source text. They are the following:

EQU	Assign a permanent value to a symbol
SET	Assign a temporary value to a symbol
REG	Assign an identifier to a processor register list
FREG	Assign an identifier to an MC68881 register list
OPWORD	Assign an identifier to an opcode

EQU and SET: Name constants and registers

name EQU { *arith-expr* | *reg* | *import-name* }

name SET { *arith-expr* | *reg* | *import-name* }

EQU and SET assign the value in the operand field to the identifier in the label field. Both fields are required. These directives are collectively called **equates.** The operand may be a numeric expression, a register name, or an identifier imported from another module.

EQU assigns a permanent value; once an identifier has been used in an EQU directive it may not be redefined in another EQU directive within its scope. SET assigns a temporary value; the same identifier may be redefined with another SET directive.

Within the scope of an equate, you can use the identifier in its label field anyplace you could have used the identifier or expression in its operand field. The results will be equivalent.

When you use EQU or SET with a numeric expression, follow these rules:

- The numeric expression *arith-expr* must not contain any forward or undefined references.
- Relocatable expressions are allowed only inside code modules and data modules.
- Equates defined outside modules or inside code modules may take any value. If you use an equate in a template or data module, its value must be in the range –32768..+32767 (that is, a signed 16-bit value).

You can equate an identifier to a floating-point constant or to any identifier previously equated to a floating-point constant. Because such constants are not evaluated until used (by a DC directive or an MC68881 machine instruction), EQU and SET store their values as strings and do not validate them.

Equates to absolute values (constants and registers) must appear in your source text before you use the equated identifiers. When the Assembler encounters a symbol in an effective address in a code module, it searches for its value first in the code module's local symbol table (if the symbol has been defined), then in the global symbol table. This means that effective addresses may not contain forward references to equates defining absolute values or registers. Forward references to relocatable equated values (for example, equates to the location-counter value) are allowed, since the Assembler always assumes that forward references refer to relocatable objects.

Warning:

If you give a forward-referenced local label the same identifier as a global absolute equate symbol, the Assembler will use the value of the global symbol and issue a name conflict warning. This is because the local identifier is not yet defined.

Here are some examples of valid equates:

```
Length      EQU     *-Start         ; Define Length, Start to location counter
Cr          EQU     $0D             ; Define the return character
X           SET     Y+10            ; Define X as the value of Y+10
X           SET     Y+20            ; Redefine X as the value of Y+20
StkPtr      EQU     A7              ; Define StkPtr as register A7
ProgCtr     EQU     PC              ; Define ProgCtr as the program counter
SuppA2      SET     ZA2             ; Define SuppA2 as a zero-suppressed A2
Alpha       EQU     (a+b)*10        ; Define Alpha with the expression's value
Pi          EQU     "3.14159"       ; Define Pi as a floating-point constant
```

REG and FREG: Name register lists

name	REG	rlist
name	FREG	fp-rlist

The REG and FREG directives assign the register list *rlist* or *fp-rlist* to the specified *name*. Lists named by REG are used with MOVEM instructions; lists named by FREG are used with FMOVEM instructions. You can use FREG only if the MC68881 directive is in force. Simple register lists are composed as follows:

- *Rm–Rn* designates registers *Rm* through *Rn* (where $m \leq n$, and *Rm* and *Rn* are both A registers or both D registers).

- *Ri/Rj/Rk...* designates registers *Ri, Rj, Rk...* where each term is an A register, a D register, or a range *Rm–Rn*.
- FP*m*–FP*n* designates registers FP*m* through FP*n* ($m \leq n$).
- FP*i*/FP*j*/FP*k*... designates registers FP*i*, FP*j*, FP*k*.... Each term is either a single register FP*n* or a range FP*m*–FP*n*.
- FPCR/FPSR/FPIAR designates the floating-point control registers FPCR, FPSR, and FPIAR, in any order. You cannot combine these registers with other registers in a register list.

Here are some examples:

D0-D1/A3	means	D0, D1, and A3
D2-D4/A1-A2/D7	means	D2, D3, D4, A1, A2, and D7
FP0-FP3/FP7	means	FP0, FP1, FP2, FP3, and FP7
FPCR/FPSR	means	control registers FPCR and FPSR

The scope and search rules for register-list identifiers are exactly the same as for equate identifiers, discussed above under "EQU and SET."

You can use identifiers defined by REG and FREG to build up more complex register lists. To do this, you concatenate them with register-lists or other register list identifiers. For example:

```
VolatileDs      REG     D0-D2                       ; Volatile D registers
VolatileAs      REG     A0-A1                       ; Volatile A registers
VolatileRegs    REG     VolatileAs/VolatileDs       ; Volatile A and D registers
ActiveRegs      REG     VolatileRegs/D6-D7/A4       ; All required registers
```

In this example, the register list ActiveRegs is defined so that it is equivalent to the simple list D0-D2/D6-D7/A0-A1/A4.

Here is a sample program fragment that shows REG and FREG directives used with MOVEM and FMOVEM statements:

```
PascalRegs.   REG       D2-D7/A3-A5         ; Names Pascal registers
FPRegs        FREG      FP0-FP7             ; Names FP registers
              - - -
P             PROC      EXPORT
              LINK      A6,#-LocalSize
              MOVEM.L   PascalRegs,-(A7)    ; Save Pascal registers
              FMOVEM.X  FPRegs,-(A7)        ; Save FP registers
              - - -
              FMOVEM.X  (A7)+,FPRegs        ; Restore FP registers
              MOVEM.L   (A7)+,PascalRegs    ; Restore Pascal registers
              RTS
              ENDPROC
```

OPWORD: Name machine instructions

name OPWORD abs-expr

OPWORD is used to assign a numeric value to the identifier *name* so that it may subsequently be used as a machine instruction. The expression *abs-expr* must have an absolute value in the range –32768..+65535 and may not contain any forward, undefined, or imported references. Identifiers defined by OPWORD may be used only inside code modules.

When the Assembler processes any mnemonic, it searches the following lists in the order shown:

1. standard opcodes and directives, including coprocessor instructions
2. macro identifiers
3. OPWORD names in the code module's local symbol table
4. OPWORD names in the global symbol table

Although the Assembler makes no assumptions about the use of OPWORD definitions, the intent of OPWORD is to allow you to define the Macintosh trap values. For example, you could define _Read as an identifier for the Macintosh File Manager read trap ($A002) as follows:

```
_Read       OPWORD      $A002       ; Define read trap call
```

To generate the value represented by an OPWORD name, use the name just like an Assembler mnemonic or macro call. For example, after the OPWORD directive just illustrated, the following will cause a machine instruction $A002 to be generated:

```
            _Read                   ; Generate $A002 trap call
```

Names defined by OPWORD may be used with parameters. The general syntax for the use of an OPWORD name is

[*label* | *macro-label*] *opword-name* [*abs-expr*],...

The expressions *abs-expr* must have absolute values in the range –32768..+65535 and must not contain any forward, undefined, or imported references. Each value is combined under logical OR with the value of *opword-name* to produce the final generated machine instruction code.

Hence the earlier example could be extended by means of the following equate:

```
Async       EQU         $400        ; Defines "async" bit for File Manager traps
```

The original _Read statement with a parameter would then generate $A402:

```
            _Read       Async       ; Generates $A402 trap call
```

OPWORD parameters must be separated by commas, but there need not be any expressions between commas. Two adjacent commas delimit an expression that does not affect the generated instruction. Hence the following statement also generates $A402:

```
_Read       ,Async,,,  ; Generates $A402
```

❖ Note: The standard Macintosh trap macros, discussed under "Macintosh Libraries" in Chapter 1, consist largely of OPWORD directive statements.

Data definitions

The data-definition and storage-allocation directives described in this section let you define constants, initialize data, and reserve storage areas in code modules, data modules, and templates. They are the following:

DC	Place constants in code or data
DCB	Place a block of constants in code or data
DS	Define a storage area

DC and DCB: Place constants in code or data

[{ *label* | *macro-label* }] DC[.*size*] { *expr* | *string* },...

[{ *label* | *macro-label* }] DCB[.*size*] *length*,{ *expr* | *string* }

DC and DCB place data in the current (code or data) module. When used outside an existing module, they define a new data module containing the specified data. The optional qualifier *size*, which is separated from the directive name by a period, consists of a letter that indicates the size of each data increment. Word (W) is the default value if you do not include the qualifier. *Size* also determines the size of the increments specified by the integer expression *length*, as shown in Table 4-1.

Table 4-1
DC and DCB data increments

Qualifier	Name	*Length* increments, in bytes
B	Byte	1
W	Word	2
L	Long word	4
S	Single precision*	4
D	Double precision*	8
X	Extended*	12
P	Packed BCD*	12

* Can be used only if the MC68881 directive is in force and floating-point constants are being defined

The operand field of a DC directive statement may contain up to 25 values, numeric expressions, and strings in any mixture, separated by commas.

The operand field of a DCB directive statement begins with a length expression that specifies the number of data increments in the data block. The size of each increment is determined by the *size* qualifier, as shown in Table 4-1. This is followed by a single value to be placed in each such increment. Hence a DCB directive statement with a *length* of *n* acts the same as *n* DC directives. The DCB *length* parameter must be an absolute expression with a value greater than 0, and may not contain any forward, undefined, or imported references.

All the values specified in a single DC or DCB directive statement make up one data module if the statement is used outside a code or data module. All the values make up one block of ascending bytes if it is used inside a code or data module. This is true even when the data module is declared as having a decrementing location counter.

All data sizes except byte (B) are aligned to the next word boundary unless an ALIGN 0 directive is in force. The optional *label* is associated with the first byte of data after alignment.

Integer expressions must fit into the size specified by the DC or DCB *Size* qualifier. For example, a value of 1000 cannot be used with a DC.B directive. Strings are formatted according to the current STRING directive setting. When used with a string value, the DC or DCB *size* qualifier affects alignment only.

Here are some examples of *size* qualifiers and data values:

```
DC.B        'Nebur L. Ari'      ; A 12-character string in current format
DC.L        1, 2, 3             ; Three long words containing 1, 2, and 3
DC.B        T1-T2               ; A byte with two relocatable references
DC.B        $FDF                ; An error ($FDF is too big for a byte)
DC.W        1                   ; A word constant containing integer 1
DC.X        "1.234"             ; A 12-byte extended constant
DC.D        "Nan(1)"            ; An 8-byte double-precision constant
DC.L        '1234'              ; A 4-character string in current format
DC.L        ('1234')            ; A 4-byte constant $31323334
```

The last example is a four-character string enclosed in parentheses. Because both strings and integer expressions may be used as DC or DCB operands, the Assembler decides the operand's type by examining its first symbol. The parentheses force the Assembler to type the operand as an integer expression. As such, it can contain a string constant of up to four characters without exceeding thelong-word size set by the L qualifier; the Assembler treats it as a right-justified 32-bit value padded on the left with zeros. In the next-to-last example, the Assembler treats the operand as an ordinary string constant.

You must take care when using DC and DCB with imported data parameters. If the current DATAREFS setting is ABSOLUTE (the default value), then any imported data reference is treated as a 32-bit absolute address, requiring a qualifier of L. Other situations require different qualifiers. For further information see "Linker and Scope Controls," later in this chapter.

When you use DC or DCB to place data in a data module, you must link your finished program with the library file Runtime.o, which contains the data initialization routine _DataInit. If your main code module is written in assembly language, its first executable statement must be a call (JSR) to the entry point _DataInit. This entry point must also be declared as IMPORT. After returning from _DataInit, your program may unload the segment %A5Init that contains it, by calling the Macintosh routine UnloadSeg. If your main program is written in C or Pascal, no explicit call to _DataInit is required, because the run-time libraries for C and Pascal automatically take care of data initialization.

DS: Define storage area

[{ *label* | *macro-label* }] DS[.*size*] { *length* / *template-name* }

The DS directive allocates and defines an uninitialized storage area in a code module, data module, or template. When used outside an existing module, it defines a new data module of the specifierd length. The optional qualifier *size*, which is separated from the directive name by a period, consists of a letter that indicates the size of each of the data increments defined by *length*, as shown above in Table 4-1. Word (W) is the default value if you do not include the qualifier. *Length* must be an absolute expression with a value greater than or equal to zero. It cannot contain any forward, undefined, or imported references.

All data sizes except byte (B) are aligned to the next word boundary unless an ALIGN 0 directive is in force. The optional label is associated with the first byte of data after alignment.

A DS directive with a *length* of 0 acts exactly like an ALIGN directive with an operand value equal to the size of the DS directive's qualifier. In this form, it ignores any prior ALIGN 0 directive. ALIGN is discussed later in this chapter under "Location-Counter Controls."

The storage area allocated by DS can also be specified by a template identifier that has been previously defined. Template definitions are discussed in the next section. In this case, the length allocated is determined by the size of the template. If you use a *label* with DS and a template identifier, that *label* is given the type represented by the template. You can then access the fields of the template by qualifying its field identifiers with the DS *label* instead of the template identifier, using the form *DSlabel.fieldname*. You can also use DS in the same way to type fields of templates and then access fields within them, using the form *DSlabel.fieldname.innerfield*. You can create nested fields in this way to any depth.

You can use template identifiers to specify DS data types in all cases except when DS is used in the code section of a code module or when there is no label specified. Although the data area allocated by DS is not typed in these cases, its size is still determined by the template's size. When you use a template identifier to specify size and/or type in a DS directive statement, that identifier must be the directive's only operand. Using the identifier any other way (such as by enclosing it in parentheses or including it in an expression) refers to the identifier's offset value instead of to its type and size.

Template definitions

A **template** describes the layout of a collection of data without actually allocating any memory space. This section describes the following template definition directives:

RECORD	Begin a record template definition
ENDR	End a record template definition
WITH	Begin default record identifier qualification
ENDWITH	End default record identifier qualification

A template definition starts with a RECORD directive statement and ends with an ENDR directive. In between are directives that describe the layout of the template, using DS, ORG, ALIGN, EQU, and SET. Sections of data within a template are called **fields.** Fields are referenced by the form *record.field*, where *record* is the template identifier and *field* is the label in the directive that defined the field. As a convenience, you may use the WITH and ENDWITH directives to specify a template identifier over a section of your source text.

RECORD and ENDR: Define a template

name RECORD { offset | IMPORT | {origin} } [, { INCR[EMENT] | DECR[EMENT] }]

DS, ORG, ALIGN, EQU, and SET directive statements

[*macro-label*] ENDR

RECORD and ENDR delimit the section of source text in which you define a template. Notice that RECORD and ENDR are also used to define data modules, as described above under "Code and Data Module Definitions." The Assembler distinguishes the two usages by the parameters in the RECORD directive statement. When used to define a data module, RECORD has either no parameters or one of the terminal symbols EXPORT or ENTRY as its first parameter. When used to define a template, RECORD always has at least one parameter—the absolute expression *offset*, the terminal symbol IMPORT, or the identifier *origin* enclosed in braces.

The definition of a template is equivalent to a sequence of equates. However, it is a more natural way to specify a storage layout. Templates may be defined only outside modules or as local definitions inside code modules.

As with code and data modules, templates have their own location counter corresponding to the next available data location. As each data field is defined, the location counter is incremented by the size of that data field. The next piece of data is then placed at the next available location. Location-counter values must be in the range –32768..+32767.

To define field locations in both positive and negative directions, you can include INCREMENT or DECREMENT in the RECORD directive, preceded by a comma. INCREMENT is the default parameter; it makes RECORD allocate fields at ascending locations, as just described. If you specify DECREMENT, fields are located at descending locations, corresponding to Pascal memory layouts. Before defining each field, the Assembler decrements the location counter by its size; hence each field starts at an address lower than the one before it.

The parameter *offset* represents an initial offset for the template. It must be an absolute expression without any forward, undefined, or imported references. Specifying a nonzero offset is equivalent to specifying a zero offset and placing an ORG directive at the start of the template definition. For example,

```
Name    RECORD    100
        - - -
        ENDR
```

defines the same template as

```
Name    RECORD    0
        ORG       *+100
        - - -
        ENDR
```

The main advantage of the specification on the left is that the template name takes the value of the initial offset. The template name can then be used in place of the offset value in arithmetic expressions.

You can specify a negative offset with RECORD. This is useful for mapping Macintosh Pascal stack frames. Suppose, for example, you want to write an external Pascal procedure Px with the following declaration:

```
PROCEDURE Px(a,b,c: INTEGER); EXTERNAL;
```

If the Pascal program calls this procedure in the form Px(a,b,c), then the following equivalent code actions will be generated by Pascal:

```
            MOVE.W      a(A6),-(A7)             ; Push a
            MOVE.W      b(A6),-(A7)             ; Push b
            MOVE.W      c(A6),-(A7)             ; Push c
            JSR         Px                      ; Call external procedure Px
```

In assembly-language procedure Px, we want to reserve stack space for local variables. So, following the conventions used by the MPW Pascal compiler, we start the subroutine with LINK A6 to reserve the local stack space, as follows:

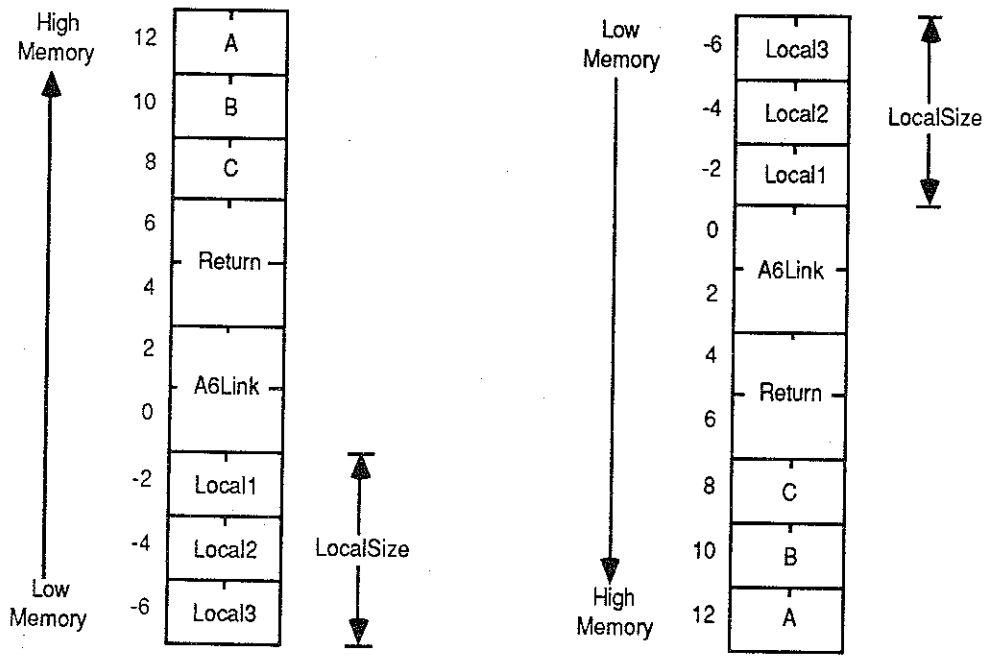
```
Px          PROC        EXPORT
            LINK        A6,#LocalSize           ; Reserve space for locals on stack
            - - -
            RTS
            END
```

We can now define the stack frame, using RECORD to delimit the following template definition:

```
StackFrame  RECORD      -6                      ; Start at -6 for 3 local integers
Local3      DS.W        1                       ; Third local
Local2      DS.W        1                       ; Second local
Local1      DS.W        1                       ; First local
LocalSize   EQU         Local3-*                ; Local area (-6) used in LINK
A6Link      DS.L        1                       ; Old value of A6 set by LINK
Return      DS.L        1                       ; Return address for RTS
C           DS.W        1                       ; c parameter
B           DS.W        1                       ; b parameter
A           DS.W        1                       ; a parameter
            ENDR
```

Figure 4-1 shows the stack frame after the LINK A6 instruction in the example has been executed. It illustrates two ways to view the same template layout.

Figure 4-1
Stack frame example



The low-to-high layout on the right matches the record template definition just given. This is because the RECORD directive assumed the default parameter INCREMENT, even though it contained a negative initial offset. By specifying DECREMENT, we could equally well define the template to match the high-to-low diagram on the left:

```
StackFrame  RECORD      14,DECR
A           DS.W        1
B           DS.W        1
C           DS.W        1
Return      DS.L        1
A6Link      DS.L        1
Local1      DS.W        1
Local2      DS.W        1
Local3      DS.W        1
LocalSize   EQU         *
            ENDR
```

```
; Start at 14 and decrement
; a parameter (at location 12)
; b parameter (at location 10)
; c parameter (at location 8)
; Return addr for RTS (location 4)
; Old A6 value set by LINK (loc 0)
; First local (at location -2)
; Second local (at location -4)
; Third local (at location -6)
; Local area (-6) used in LINK
```

Notice that in both of the foregoing stack frame layouts the initial offset had to be given. For most mappings the offset will be 0. However, for stack frames the initial offset must be chosen so that the A6Link field will have an offset of 0. This is why we specified -6 in the sample incrementing layout and 14 in the sample decrementing layout. It is not necessary, however, to compute the size of each template and enter it as an absolute value. You can use the {*origin*} parameter to make the Assembler do this work for you. The template origin is defined as the field which is to have an offset of 0. You specify a field identifier enclosed in braces to indicate that that field is to be the template's origin. The Assembler then reads in the the template definition as if the initial offset was 0 (displaying these values in the Assembly listing) and subtracts the zero-relative offset of the origin field from each field offset. The effect is to shift the template's origin from the start of the template to the field specified by the origin parameter.

Hence in the foregoing examples we could have used simpler RECORD directive forms, leaving the rest of the template definitions unchanged:

```
StackFrame          RECORD          {A6Link}

StackFrame          RECORD          {A6Link},DECR
```

Notice that shifting the origin of a template affects only the field offsets. Equates are not changed. In origin-shifted templates, the Assembler distinguishes between equates to absolute expressions (such as Local3-* in the incrementing example) and equates to other field identifiers.

Normally you define some dynamic data, such as the stack frame illustrated above, and then use a template to map over the data. However, you may also have static data, defined somewhere else in your assembly-language program as a data module, that you want to map. Because both templates and data modules are defined by RECORD directives, they have the same underlying form. This lets you import an entire data module and directly access its fields. You map a data module by specifying its identifier in an IMPORT directive and then using that same identifier as a template label in a RECORD directive. Alternatively, you can specify the IMPORT directive identifier explicitly as the RECORD template parameter. The base register for a data module imported as a template is always A5.

In C programming, the situation just described corresponds to declaring a static structure (struct) as external, and then importing the entire structure (extern) from another file. The struct declaration must appear in both files, just as the RECORD directive appears twice in the assembly-language source text.

Using templates as data types

In higher-level languages, data types define the specific ways that data is stored in memory. For example, a Pascal record type or a C struct type specifies the memory layout and size of all data items of that type. Fields within data structures also have types; hence higher-level languages allow the creation of complex structures of typed data.

In the MPW assembly language, templates serve the same purpose. To create the equivalent of a data type, you use the identifier of a template (the label you used in the RECORD directive that created it) as the operand of a DS directive. For example:

```
label       DS          template-name
```

Specifying a template identifier alone makes the DS directive allocate an amount of memory equal to the size of the template. If a label is also specified, that label acquires the type represented by the specified template. You can then access fields of the template by qualifying the field identifiers with the DS label instead of the template identifier, in the form *label.fieldname*. Fields of templates can themselves be typed the same way. You can identify them by a series of qualifications, in the form *label.fieldname.innername*. Fields can be nested this way to any depth.

The following is an example of how template types are used. The example shows the Macintosh QuickDraw definitions for points and rectangles. The corresponding Pascal type declarations are shown as comments:

```
Point       RECORD    0       Point = RECORD CASE INTEGER OF
v           DS.W      1                 0: (v: INTEGER;
h           DS.W      1                    h: INTEGER);
            ORG       v
vh          DS.W      2                 1: (vh: ARRAY [VHSelect] OF INTEGER)
            ENDR                      END;

Rect        RECORD    0       Rect = RECORD CASE INTEGER OF
top         DS.W      1                 0: (top: INTEGER;
left        DS.W      1                    left: INTEGER;
bottom      DS.W      1                    bottom: INTEGER;
right       DS.W      1                    right: INTEGER);
            ORG       top
topLeft     DS        Point             1: (topLeft:  Point;
botRight    DS        Point                botRight: Point)
            ENDR                      END;
```

In this example, both `topLeft` and `botRight` are defined as having type `Point`. `Point` and `Rect` may now be used to allocate space in a data module. For example:

```
MyData      RECORD                              ; Define a data module
            - - -
MousePt     DS          Point
DragRect    DS          Rect
            - - -
            ENDR
```

We can now access the various fields of MousePt and DragRect by using the field identifiers established in the template definition. Because these fields are in a data module, the base register is A5. Here are some examples:

```
MOVE.W      MousePt.v(A5),D0                ; Get v component
MOVE.L      MousePt.vh(A5),D0               ; Get full point position
MOVE.W      DragRect.left(A5),D0            ; Get left coordinate
MOVE.L      DragRect.topLeft(A5),D0         ; Get topLeft of rectangle
MOVE.W      DragRect.botRight.h(A5),D0      ; Get botRight h component
```

You can use templates as types in DS directives any time except when you use DS in the code section of a code module, or when the DS directive has no label. Although the Assembler does not establish a type in those cases, it still uses the template size to define the space allocated by DS. To use a template identifier as a size and/or type specification, you must supply it as the only operand in the DS directive statement. Using the identifier any other way (for instance, enclosing the identifier in parentheses or using the identifier in an expression) makes DS use the template's value instead of its identifier.

In the foregoing example, incrementing templates were used to define types. Decrementing templates may also be used. If you use a decrementing template as a type, its origin for data allocation purposes is shifted so that its lowest address corresponds to a location counter value of 0. You can freely mix incrementing and decrementing templates to define complex data types.

WITH and ENDWITH: Supply RECORD name qualification

[*macro-label*]	WITH	*name,...*
	Code-module statements	
[*macro-label*]	ENDWITH	

The WITH directive lets you access RECORD field identifiers without explicit qualification. WITH may only be used inside code modules (modules delimited by PROC, FUNC, or MAIN). You can write a series of identifiers, separated by commas, as parameters; they all become field qualifiers. They may be the identifiers of data modules, templates, or typed fields. For a description of field typing see "Using Templates as Data Types," above. The implicit qualification established by WITH remains in effect until a matching ENDWITH or the end of the code module.

Using the StackFrame RECORD template illustrated earlier, the following example shows how WITH can be used to access a template's fields:

```
LINK    A6,#StackFrame.LocalSize
MOVE    StackFrame.A(A6),D0
MOVE    StackFrame.B(A6),D1
MOVE    StackFrame.C(A6),D2
- - -
```

is equivalent to

```
WITH    StackFrame
LINK    A6,#LocalSize
MOVE    A(A6),D0
MOVE    B(A6),D1
MOVE    C(A6),D2
- - -
ENDWITH
```

WITH directives may be nested. Alternatively, more than one identifier may be specified in a single WITH directive. The latter is equivalent to nesting WITH directives, with the last parameter being considered the most deeply nested. For example,

```
WITH    alpha,beta,gamma
- - -
ENDWITH
```

is equivalent to

```
WITH  alpha
    WITH  beta
        WITH  gamma
          - - -
        ENDWITH
    ENDWITH
ENDWITH
```

You can nest field qualifications, using WITH directives in either form. Parameters occurring earlier will qualify parameters occurring later. The Assembler searches for each specified field identifier, starting with the most deeply nested WITH, and attaches the qualification when it finds it. If two fields have identical identifiers, it supplies the most deeply nested qualification. Here is an example, based on the DragRect definition given above under "Using Templates as Data Types" in the discussion of REC and ENDREC:

```
WITH        DragRect,topLeft      ; Qualify with DragRect and DragRect.topLeft
MOVE.W      v(A5),D0              ; DragRect.topLeft.v
MOVE.W      left(A5),D2           ; DragRect.left
MOVE.W      botRight.h(A5),D1     ; DragRect.botRight.h
ENDWITH
```

Notice here that topLeft is subject to the WITH qualification of DragRect, the WITH directive's first parameter. This is equivalent to the explicit qualification DragRect.topLeft. The v field reference is qualified by topLeft, so that it is equivalent to a reference to DragRect.topLeft.v. The left field is a field of DragRect, so that it is equivalent to a reference to DragRect.left. The last reference is to the h field of botRight, which is itself a field of DragRect. However, h also occurs as a field identifier in topLeft. The explicit reference to botRight is required to override the implicit qualification topLeft.

As you can see from the last example, nested WITH directives can generate unintended field identifier qualifications, leading to very subtle program bugs. We recommend that you use WITH only to cover short sections of source text, and avoid complex nestings. If in doubt, replace WITH directives with fully qualified field identifiers. Your program will become easier to understand and maintain when the reader always knows to which module or template every field belongs.

Linker and scope controls

The directives described in this section all pass information to the MPW Linker. They tell the Linker how to associate identifiers between object files, how to group individual modules into segments, and how to comment the object-code file. At the same time, they give the Assembler information about the scope of objects named in the directives and tell it whether they are code or data. They are the following:

EXPORT	Make entry points accessible in other assemblies
ENTRY	Make local entry-points global
IMPORT	Identify entry points declared externally
CODEREFS	Control the linking of code-to-code references
DATAREFS	Control the linking of data-to-code and data-to-data references
SEG	Specify the current code segment
COMMENT	Place a comment in the object file

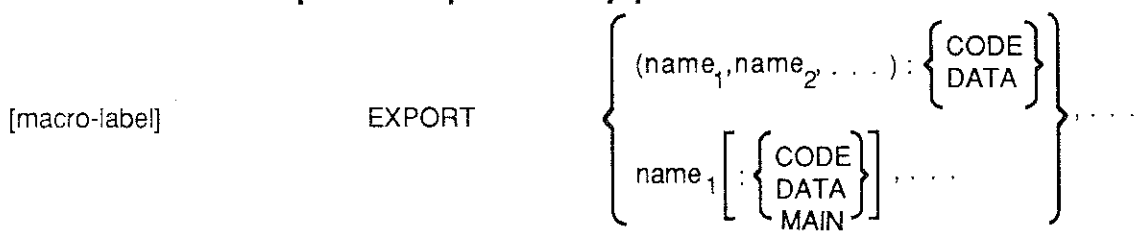
ENTRY, EXPORT, and IMPORT all affect the scope of code or data module identifiers. ENTRY promotes an identifier to global scope, so that it is accessible to all references in the same file. EXPORT has the same effect as ENTRY; in addition, it makes the identifier accessible to other files. IMPORT provides a reference in the current module or assembly for identifiers exported in another module, assembly, or compilation.

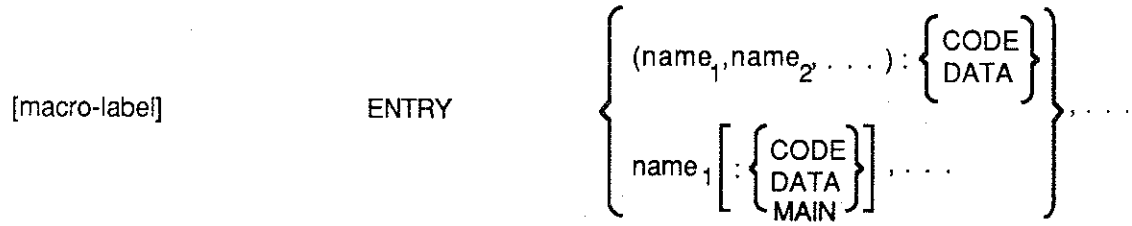
CODEREFS and DATAREFS allow you to control some of the characteristics of the way the Linker associates code and data references created by EXPORT, ENTRY, and IMPORT.

SEG specifies the code modules in a segment. For a discussion of code module segments, see "Segmentation" in Chapter 2.

COMMENT tells the Linker to generate a comment record for your object file.

EXPORT and ENTRY: Expand scope of entry points





With the exception of local labels, all identifiers in a code or data module are automatically accessible throughout the module in which they are defined. EXPORT and ENTRY extend the scope of specified identifiers by making them accessible in other modules as well. EXPORT makes them accessible in modules in all files linked to the file containing it; ENTRY makes them accessible only in modules within the same assembly.

Identifiers listed in EXPORT or ENTRY directives are said to be "exported." Each one must be designated as Code or Data; one may be designated as MAIN. You can accept the default designations or you can specify them explicitly. These rules govern how the Assembler treats the operands of EXPORT and ENTRY directives by default:

- The default designation for identifiers listed inside a code module is Code.
- The default designation for identifiers listed inside a data module is Data.
- The default designation for identifiers listed outside any module is Code.

You can override any of these default designations by writing explicit declarations. If you specify CODE, DATA, or MAIN explicitly, you can either write a series of identifiers separated by commas and enclosed in parentheses, followed by a declaration:

EXPORT *(name1,name2,name3)*:DATA

or you can write a separate declaration for each name, omitting the parentheses:

ENTRY *name1*:DATA,*name2*:CODE,*name3*:DATA

Here are the rules for using EXPORT and ENTRY:

- You must place each EXPORT or ENTRY directive in your source text before defining any of the identifiers it affects.
- The directive must be written within the existing scope of all identifiers it affects.
- An exported identifier may not be identical to any other identifier within its new scope.
- You export a module identifier either by using EXPORT or ENTRY before the directive that starts the module (PROC, FUNC, MAIN, or RECORD) or by including EXPORT or ENTRY in the module directive's parameter list.
- You export identifiers occurring within a module by including an EXPORT or ENTRY directive inside the module, before they are defined.

- An identifier may be mentioned in more than one EXPORT or ENTRY directive, provided it is not listed as both code and data.
- An EXPORT directive mentioning an identifier previously listed in an ENTRY directive supersedes the ENTRY.
- An ENTRY directive mentioning an identifier previously listed in an EXPORT directive has no effect.
- An identifier designated as CODE in an ENTRY or EXPORT directive may be later designated as MAIN by an ENTRY, EXPORT, or MAIN directive.
- Only one identifier in an assembly may be designated as MAIN.
- You cannot include a qualification when exporting a field identifier. Only the unqualified field identifier will be exported.

The following example illustrates the placement of EXPORT statements in a file:

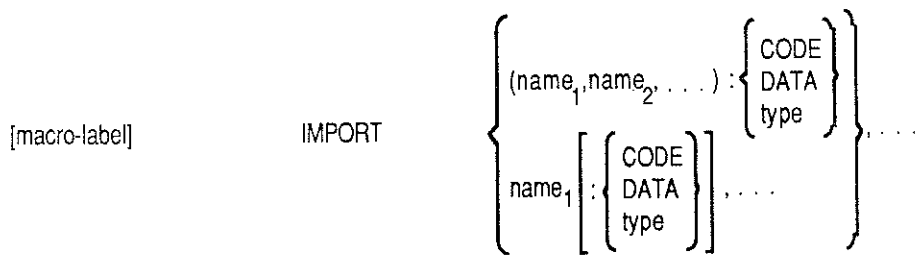
```
            EXPORT      X                   ; Export code module name X
            EXPORT      Y:DATA              ; Export data module name Y
Y           RECORD                          ; Could have exported from here
            EXPORT      Field1,Field2       ; Export inside module
Field1      DS.W        1
Field2      DS.L        2
            ENDR

X           PROC                            ; Could have exported from here
            EXPORT      Z                   ; Declare secondary entry point
            - - -                           ; code for X

Z                                           ; Secondary entry point Z

            - - -                           ; more code for X
            END
```

IMPORT: Identify external entry points



The IMPORT directive makes specified identifiers accessible to the file or module in which it occurs. Such identifiers are said to be "imported." Every imported identifier must be declared elsewhere in one of the following ways:

- as either ENTRY or EXPORT in other modules of the same assembly
- as EXPORT in other assemblies
- as an exported procedure, function, or global variable in another language

In the syntax diagram above, *type* is the identifier of a template used to define a record structure, as explained in the discussion of RECORD and ENDR under "Template definitions," above. As explained there, the template itself may be declared as IMPORT. Using an IMPORT directive, however, lets you import several templates under other identifiers without having to modify the original template declarations.

Here are the principal rules governing the use of the IMPORT directive:

- The inclusion of an identifier in an IMPORT statement is treated as a definition of the identifier with respect to identifier scope. This means that imported identifiers follow the standard local/global scope rules covered in Chapter 2 under "Scope of Definitions."
- Imported identifiers that are to be made accessible to more than one module in a file must be imported before any modules that use the identifiers are defined.
- Imported identifiers local to a module must be listed in an IMPORT directive inside that module before they are mentioned in any other statement.
- The Assembler does not verify the CODE or DATA designation of imported identifiers. Hence you should give them the same designation they had when they were exported.
- You can access fields of imported templates by qualifying the identifiers used in IMPORT with the original field identifiers.
- Imported code identifiers may be used in all PC-relative effective address modes. However, their use in short branches and in indexed modes with 8-bit displacements may result in run-time errors.

The Assembler gives default CODE or DATA assignments to the operands of the IMPORT directive according to these rules:

- The default designation for identifiers listed inside a code module is CODE.
- The default designation for identifiers listed inside a data module is DATA.
- The default designation for identifiers listed outside any module is CODE.

You can override any of these default designations by writing CODE or DATA explicitly, as described above under "EXPORT and ENTRY."

Referring to the example given above under "EXPORT and ENTRY," the following example shows how the identifiers exported there could be imported into another file. It also illustrates the identifier scope rules:

```
            IMPORT      X                       ; Import X as a code identifier
            IMPORT      (Field1,Y):DATA         ; Import Field1 and Y as data
W           PROC
            IMPORT      (Z,L1):CODE             ; Allow local access to Z and L1
            IMPORT      Field2:DATA             ; Allow local access to Field2
            - - -
            MOVE.L      Field2,D1               ; Field2 accessible only from module W
            JSR         L1                      ; Call L1 in another module
            MOVE.W      D0,Field1               ; Copy D0 into Field1 (in other file)
            - - -
            ENDPROC
```

CODEREFS and DATAREFS: Control name linking

[macro-label] CODEREFS { F[ORCE[JT]] | NOF[ORCE[JT]] | F[ORCE]PC }

[macro-label] DATAREFS { R[EL[ATIVE]] | A[BS[OLUTE]] }

CODEREFS lets you control how the MPW Linker treats code-to-code identifier references—that is, references from one code module to another. With NOFORCEJT, a reference to an address in the same segment will cause the Linker to treat it as PC-relative. References between segments will go through the jump table. With FORCEJT, all references will go through the jump table even if they are in the same segment. FORCEPC is the inverse of FORCEJT; it requires that all code-to-code references be PC-relative and in the same segment, and causes a Linker error if any are not. CODEREFS NOFORCEJT is the preset condition.

DATAREFS lets you control how the Linker treats data-to-code and data-to-data identifier references. With ABSOLUTE, all references to code or data are 32-bit absolute jump-table addresses and may be used only in DC.L statements. With RELATIVE, all references to code are A5-relative jump-table offsets and all references to data are A5-relative offsets. You may use DC.W and DC.L for RELATIVE references. DATAREFS ABSOLUTE is the preset condition.

Code-to-data identifier references are always A5-relative; they are unaffected by CODEREFS or DATAREFS.

You can write operands for CODEREFS and DATAREFS in any of the following alternate forms:

- FORCEJT as FORCE or F
- NOFORCEJT as NOFORCE or NOF
- FORCEPC as FPC

- RELATIVE as REL or R
- ABSOLUTE as ABS or A

To understand the operation of CODEREFS and DATAREFS fully, you must know their effects on the linking process. The four possible identifier reference combinations between code and data modules are discussed below.

Code-to-Code references

When a reference points from one code module to another and CODEREFS NOFORCEJT is in effect, the Linker checks to see whether both code modules belong to the same segment. If so, it changes the addressing mode to PC-relative with a 16-bit displacement and sets the appropriate displacement. If the reference is to a code location in a different segment, the Linker converts the address to a location in the jump table (a positive offset from A5). The Linker assumes that the word immediately before the 16-bit displacement represents an instruction which has its destination mode and register fields in bits 0 through 5 (the six low-order bits). The Linker does not support 32-bit instruction editing caused by MC68020 addressing modes.

The Linker accepts all code references from one module to another. With instructions other than JSR, JMP, PEA, and LEA, however, the referenced identifiers must be in the same segment.

To summarize, the Linker follows these rules when checking for illegal code-to-code references:

- If the reference is to a module in the same segment, the Linker accepts it for any instruction and any CODEREFS setting. If the instruction being used is not JSR, JMP, PEA, or LEA, the referenced identifier must be in the same segment.
- If the reference is to a module in another segment, the Linker accepts it only for JSR, JMP, PEA, and LEA instructions, and only if CODEREFS FORCEPC is not in effect.
- If the reference is to a module in another segment and CODEREFS FORCEPC is in effect, the Linker will not accept it.

You can use CODEREFS FORCEJT to forestall certain run-time problems. If, for example, you want to save the PC-relative address of a procedure to call it later and that procedure belongs to a segment that may become unloaded, then attempting to call the procedure after the segment has been unloaded will not work. The solution is to use CODEREFS FORCEJT. It forces all code-to-code references to go through the jump table anyway, as if the modules were in different segments. Saving a jump table address and using it to call a procedure later guarantees that the corresponding segment will be loaded, even if it is currently unloaded.

Code-to-Data references

When a reference points from a code module to a data module, the Assembler always generates an A5 offset. If you do not need indexing and the Assembler knows the reference is to data for a machine instruction, you can omit the A5 reference; the Assembler will generate it for you.

Data-to-Code references

You can reference a code address from a data module only with a `DC` instruction. When you do this, you can choose to make the Linker generate a jump table offset or let it generate the actual run-time jump-table address. If you make the Linker generate an offset, you can specify its size as a word or a long word by qualifying the `DC` instruction (`DC.W` or `DC.L`). If you let the Linker generate the actual run-time jump-table address, you must specify `DC.L`, because a 32-bit address will be added to the `DC` statement at load time. But any DC reference to a local label in the same code module (or in a nested data module) will always be treated as a module offset.

`DATAREFS` controls the two forms of data-to-code addressing. `DATAREFS RELATIVE` indicates that offsets are to be used, while `DATAREFS ABSOLUTE` (or no directive at all) indicates that A5-relative 32-bit absolute jump-table addresses are to be generated.

Data-to-Data references

Data-to-data references are similar to data-to-code references. You can force the Linker to generate an A5-relative offset to the data by using `DATAREFS RELATIVE`, or you can let it refer to the 32-bit absolute address created at run time by doing nothing or by using `DATAREFS ABSOLUTE`.

Table 4-2 summarizes the effects of `CODEREFS` and `DATAREFS` in the four cases just discussed.

❖ *Note:* If your program uses absolute data references, you must link it with the library file Runtime.o, which contains the data initialization routine `_DataInit`. If your main code module is written in assembly language, its first executable statement must be a call (`JSR`) to the entry point `_DataInit`. This entry point must also be declared as `IMPORT`. After returning from `_DataInit`, your program may unload the segment `%A5Init` that contains it, by calling the Macintosh routine `UnloadSeg`. If your main program is written in C or Pascal, no explicit call to `_DataInit` is required, because the run-time libraries for C and Pascal automatically take care of data initialization.

Table 4-2
Effects of CODEREFS and DATAREFS

From	To	Directive	Effect
Code	Code	`CODEREFS FORCEJT`	Always uses jump table
Code	Code	`CODEREFS FORCEPC`	Always PC-relative
Code	Code	`CODEREFS NOFORCEJT`	Uses jump table if across segments
Code	Data		Always generates A5 offset
Data	Code	`DATAREFS RELATIVE`	Uses jump table (`W` or `L`) offset
Data	Code	`DATAREFS ABSOLUTE`	Uses 32-bit absolute jump-table addresses (`L`)
Data	Data	`DATAREFS RELATIVE`	Generates A5 (`W` or `L`) offset
Data	Data	`DATAREFS ABSOLUTE`	Uses 32-bit absolute addresses (`L`)

SEG: Specify Current code segment

[macro-label] SEG *[str-expr]*

All code modules are grouped into segments, as discussed in Chapter 2 under "Source Text Structure." The `SEG` directive lets you control this grouping. It declares that all subsequent code modules (ignoring any data modules) are to be placed in the segment named by the expression *str-expr*. `SEG` takes effect at the next `PROC`, `FUNC`, or `MAIN` directive. It remains in effect until the next `SEG` directive. The modules thus placed in one segment need not be contiguous in the source text.

The default value of *string* is `Main` (uppercase *M*, the rest lowercase, as shown). If you do not use the `SEG` directive or use it without an operand, all subsequent code modules will be placed in the `Main` segment.

❖ *Note:* Code segment names are case-sensitive. Be careful to use ientical capitalization when writing segment names that are to be treated as identical.

Code modules in the same segment do not have to be contiguous in the source file. Code modules belonging to other segments may be mixed with them as long as they fall under the appropriate SEG directive. Here's an example:

```
        SEG     'Name1'
A       PROC
        - - -
        ENDP
        SEG     'Name2'
B       PROC
        - - -
        ENDP
C       PROC
        - - -
        ENDP
        SEG     'Name1'
D       PROC
        - - -
        ENDP
        SEG     'Name2'
E       PROC
        - - -
        ENDP
```

In this example, modules A and D belong to the segment 'Name1'; B, C, and E belong to the segment 'Name2'.

COMMENT: Place a comment in the object file

[macro-label] COMMENT *str-expr*

The COMMENT directive lets you place a comment in your unlinked object file. It causes the Assembler to generate a object file comment record containing the value of the COMMENT directive's string expression operand.

Assembly options

The assembly option directives described in this section let you control certain assumptions the Assembler makes about the program it is assembling. The assembly option directives and the assumptions they control are as follows:

MACHINE	Identify the target microprocessor model
MC68881	Control the assembly of floating-point coprocessor instructions
MC68851	Control the assembly of PMMU coprocessor instructions
STRING	Control the encoding of string constants
BRANCH	Control the encoding of branch instructions
FORWARD	Control the encoding of forward references
OPT	Control the level of code optimization
CASE	Control the treatment of lowercase letters in identifiers
BLANKS	Control the treatment of spaces and tabs in the operand field

MACHINE: Specify target machine

[macro-label] MACHINE { MC68000 | MC68010 | MC68020 | MC68030 }

This directive tells the Assembler that the target microprocessor is an MC68000 (the preset assumption), an MC68010, an MC68020, or an MC68030. The Assembler will accept only those instructions and addressing forms supported by the target microprocessor. A MACHINE directive has effect until the end of the source text file or until another MACHINE directive is encountered.

MC68881: Assemble MC68881 coprocessor instructions

[macro-label] MC68881 *[fp-option]*,...

This directive tells the Assembler that subsequent source text may contain instructions to an MC68881 floating-point coprocessor and specifies how the Assembler is to interpret them. Hence your source text must contain this directive before the first such instruction. Each *fp-option* operand consists of a keyword, an equal sign, and an expression. There are four possible options:

COID=*expr*
PREC[ISION]={ X | D | S }
ROUND[ING]={ N | U | D | Z }
KFACTOR=*expr*

The numeric expression *expr* following COID is the coprocessor ID number of the MC68881 coprocessor, in the range 1..7. It has a default value of 1.

The letter following PRECISION indicates how much precision and range the Assembler should retain when converting floating-point constants in the source code into binary values. The default value is extended (X); however, you may alternatively specify double precision (D) or single precision (S).

The letter following ROUNDING indicates how the Assembler should round floating-point constants in the source code when converting them into binary values. The default value is to round to the nearest representation (N); however, you may alternatively specify rounding upward (U), downward (D), or toward zero (Z).

The numeric expression *expr* following KFACTOR specifies the default k-factor that the coprocessor uses when interpreting FMOVE.P instructions in which the k-factor is not explicit. The k-factor tells the MC68881 coprocessor in what format to construct the resulting decimal string. The preset default value is –16, but you can specify any value in the range –64..+63. For an explanation of k-factors, see the discussion of FMOVE in the Motorola *MC68881 Floating-Point Coprocessor User's Manual.*

Here are some rules about writing the MC68881 directive:

- If you include one or more operands, the Assembler will change only these characteristics specified by those operands from their previous values.
- If you do not include any operands, the Assembler will reset all four characteristics to their default values—COID=1, ROUNDING=N, PRECISION=X, and KFACTOR=-16.
- Operands may be written in any order.

❖ *Note:* The Macintosh ROM contains routines that perform a variety of fixed-point mathematical operations. For information about these routines, see the Toolbox Utilities chapter of *Inside Macintosh.*

MC68851: Assemble MC68851 coprocessor instructions

[*macro-label*] MC68851

This directive tells the Assembler that subsequent source text may contain instructions to an MC68851 Paged Memory Management Unit coprocessor. Your source text must contain this directive before the first such instruction. You may not use an MC68851 directive in the same assembly with a MACHINE MC68030 directive.

STRING: Specify string format

[macro-label] STRING { ASIS | PASCAL | C }

The STRING directive tells the Assembler how to encode all string constants occurring in the data-definition directives DC and DCB and in literals. The Assembler encodes strings as specified until it processes the next STRING directive. STRING PASCAL is the preset condition.

You can supply any one of these three operands with STRING:

Operand	Effect
ASIS	Strings are encoded exactly as specified; they contain just the characters included between the single quotation marks.
PASCAL	Pascal-formatted strings are generated. Each one is preceded by a length byte, as if it were stored in a Pascal variable of the type STRING.
C	C-formatted strings are generated. This format always contains at least one 0 byte following the last character of the string.

The Assembler may add one or more 0 bytes after the last character of any string, to end it on a word or long word boundary. Literals and strings defined by DC.W are filled to the next word boundary; strings defined by DC.L are filled to the next long word boundary.

BRANCH and FORWARD: Resolve Forward Branches

[macro-label] BRANCH { S[HORT] | B[YTE] | W[ORD] | L[ONG] }

[macro-label] FORWARD { W[ORD] | L[ONG] }

BRANCH and FORWARD tell the Assembler what size to assume for the displacement encodings of forward-referenced identifiers. BRANCH covers the branch instructions B*cc*, BSR, and BRA, if no size specification (S or L) is given with the mnemonic. FORWARD covers the base and outer displacements for the MC68020 extended addressing modes 6 and 73, shown in Table 3-2. Size specifications are discussed in Chapter 3 under "Forward-Reference Addressing."

BRANCH WORD and FORWARD WORD are the preset conditions; they generate 16-bit displacements. If you specify S, SHORT, B, or BYTE, the Assembler generates 8-bit displacements. If you specify L or LONG it generates 32-bit displacements. Each BRANCH or FORWARD directive remains in effect until the next BRANCH or FORWARD. You can specify FORWARD LONG with MC68020 instructions only.

❖ *Note:* The Assembler will report an error on any forward-referencing instruction with too small a displacement field. For example, if BRANCH S is specified but the Assembler generates a displacement that is too great for eight bits, the Assembler will report an error.

OPT: Specify Level of Code Optimization

[macro-label] OPT [{ALL | NONE | NOCLR}]

The Macintosh Workshop Assembler accepts generic forms for certain machine instructions and addresses, converting them to other forms at run time. The instructions for which it accepts generic forms are listed in Appendix A; the address formats are listed in Table 3-2. There are three general reasons for making such conversions:

- Optimization: The Assembler converts instructions and addresses if they can be encoded more efficiently. The result occupies less memory and often runs faster as well. An example of an instruction conversion is SUBA A*n*,A*n* in place of MOVE #0,A*n*. Examples of address conversions are *bd*(PC) for (*bd*,PC) and the suppression of MC68020 addressing base displacements and outer displacements when their values are 0.
- Convenience: The Assembler converts instructions on the basis of their context—for example, ADDI in place of ADD. It also permits substituting instructions to make coding easier and more readable—for example, by substituting BZ for BEQ.
- Compatibility: The MPW Assembler converts certain instructions to make them compatible with other assemblers. Examples include substituting BHS for BCC and BLO for BCS.

The generic address forms listed in Table 3-2 are all converted for optimization. The generic instruction forms listed in Appendix A are grouped by their reasons for conversion.

The OPT directive lets you control parts of this conversion process. You might want to eliminate optimization, for example, when writing a table of branch instructions or a routine with a critical execution time. You can specify one of three operands:

- ALL allows all conversions. This is the preset case.
- NONE eliminates all optimizations; the Assembler will not accept generic instruction forms and will not optimize addressing modes.
- NOCLR is similar to ALL. On some hardware, a CLR *ea* instruction in place of MOVE #0,*ea* is not exactly equivalent. NOCLR provides for this difference by allowing all conversions except for the MOVE-to-CLR substitution.

The current OPT directive setting remains in effect until the next OPT directive is processed.

CASE: Specify Treatment of Lowercase Letters

[macro-label] CASE { ON | Y[ES] / OFF | N[O] / OBJ[ECT] }

The CASE directive lets you determine how the Assembler interprets lowercase letters in identifiers.

ON, Y, or YES forces the Assembler to treat uppercase and lowercase letters as distinct. For example, the Assembler treats abcd and Abcd as two different identifiers. This is the way C treats uppercase and lowercase letters.

The operand OFF, N, or NO lets the Assembler treat uppercase and lowercase letters identically. For example, the Assembler treats abcd and AbCd as the same identifier. It is the preset condition.

OBJECT or OBJ forces the Assembler to generate in the object file all module identifiers and all exported and imported identifiers exactly as specified in the source text, retaining uppercase and lowercase distinctions. It ignores uppercase and lowercase distinctions for references within the source text, however. Hence CASE OBJECT has the same effect as CASE OFF within an assembly.

❖ *Note:* The CASE directive has no effect on segment names. They are always case-sensitive.

The MPW Linker always distinguishes between uppercase and lowercase when matching exported entry-point identifiers with their imported references. Hence you must be careful when linking an assembly-language program with programs written in C or Pascal. When CASE OFF is in effect, the Assembler generates exported and imported identifiers entirely in upper case. This matches the MPW Pascal Compiler, which also generates uppercase identifiers. When CASE ON is in effect, the Assembler preserves the capitalization used in the source text. This matches the MPW C Compiler, which maintains case distinctions.

CASE OBJECT lets you communicate with C in uppercase and lowercase, without needing to preserve case distinctions inside your program. Case distinctions can create a problem when you are using large files of equates—for instance, the standard Macintosh equates. CASE OBJECT lets you preserve case distinctions in your object file while ignoring them in your source text.

The CASE directive remains in force until the next CASE directive is processed. However, it is not a good idea to mix CASE modes within a single source file.

Writing register names

The MPW Assembler predefines two sets of all the register names listed in Table 3-4: one set all uppercase, one set all lowercase. With CASE OFF (the preset condition), the lowercase set of names is superfluous. With CASE ON, you can use both sets interchangeably. However, with CASE ON the Assembler does not accept register names with any case combination except all uppercase or all lowercase; register names such as Sp and zA7 are illegal. You can get around this by writing specific equates to legal register names.

BLANKS: Control acceptance of blanks in operand field

[macro-label] BLANKS { ON | Y[ES] / OFF | N[O] }

The BLANKS directive controls where the Assembler will accept spaces and tabs within the operand field. It is discussed in Chapter 2 under "Machine Instruction Syntax."

An operand of OFF, N, or NO lets the Assembler accept spaces and tabs only in places where the operand field is incomplete: following commas separating operand subfields and between paired constructs such as parentheses, brackets and braces. Where the operand field is complete, a space or a tab signals the beginning of the comment field.

An operand of ON, Y, or YES forces the Assembler to accept spaces and tabs anywhere in the operand field, except within single symbols (such as identifiers). With BLANKS ON, you must write a semicolon at the end of the operand field to separate it from the comment field. BLANKS ON is the preset condition.

The BLANKS directive remains in force until the next BLANKS directive is processed.

Location-Counter controls

The two directives described in this section control the value of the current location counter, represented in your source text by the asterisk (*) symbol. They are the following:

ALIGN	Advance the location counter to the next multiple of a value
ORG	Set the value of the location counter

ALIGN: Align location counter

[macro-label] ALIGN *[expr]*

ALIGN sets the current module or template location counter to the next multiple of the value specified by *expr*. In other words, the statement ALIGN *a-expr* acts the same as ORG *o-expr*, where *o-expr* mod *a-expr*=0. With ALIGN, *expr* usually has a value of 2 or 4, producing word or long word alignment. After processing ALIGN, the Assembler assigns memory locations to subsequent statements and directives starting with the new location-counter value. The parameter *expr* may not contain any forward, undefined, or imported references. Except for the special case of an *expr* value of 0, discussed below, ALIGN directives may appear only inside code modules, data modules, or templates.

ALIGN followed by an *expr* value of 0 tells the Assembler to perform no automatic alignments until the next ALIGN directive with a nonzero operand. In this form, an ALIGN directive may appear anywhere in your source text. It overrides the Assembler's normal process of aligning all code and all data structures larger than bytes on word boundaries. Nevertheless, the Assembler will continue to issue warning messages when locating such objects at odd addresses. You can use a DS directive to override ALIGN 0 temporarily; for further information, see the discussion of the DS directive under "Data Definitions" earlier in this chapter.

Remember these points when using ALIGN:

- When you use ALIGN in decrementing templates and data modules, the Assembler decrements the location counter. In other words, it aligns in the same direction as the prevailing counter direction.
- If you use ALIGN with no operand, the Assembler assumes a value for *expr* of 2, thereby aligning the current location counter to the next word boundary.

ORG: Set Location Counter

[*macro-label*] ORG [*expr*]

ORG sets the current module or template location counter to the value specified by the expression *expr*.

Remember these points when using ORG:

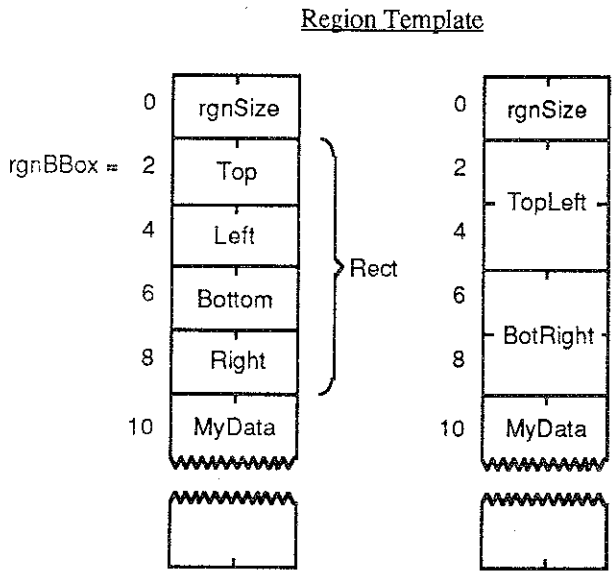
- You cannot use ORG to change the location counter from positive to negative (or vice versa) in code modules, data modules, or imported templates. You can change the sign of the location counter only in nonimported templates.
- You must be careful when using ORG to set the current location counter backward in a decrementing template. Remember that in a decrementing template each label is defined by first decrementing the location counter and then assigning its value to the label. You must take this additional decrement into account.
- If you use ORG with no operand, the Assembler sets the current location counter to the maximum positive or maximum negative location-counter value assigned to the module up to this point. It sets it to the maximum positive value for code modules and for incrementing templates and data modules. It sets it to the maximum negative value for decrementing templates and data modules.

The following example illustrates the use of ORG in a template definition:

```
Region      RECORD      0
rgnSize     DS.W        1                   ; Integer
rgnBBox     EQU         *                   ; Rect
Top         DS.W        1
Left        DS.W        1
Bottom      DS.W        1
Right       DS.W        1
            ORG         Top                 ; Points mapped over Rect
topLeft     DS.L        1
botRight    DS.L        1
            ORG                             ; Make sure of location counter
MyData      DS.B        100                 ; Reserve 100 bytes
            ENDR
```

The template just defined has the memory format shown in Figure 4-2.

Figure 4-2
Sample template format



In this example, the first ORG directive sets the location counter to `Top`, so that `topLeft` and `botRight` map onto `Top`, `Left`, `Bottom`, and `Right`. The second ORG directive has no operand, so the Assembler sets the location counter to the highest value so far—in this case, 10. Although the mapping is exact, using ORG guarantees a correct final value for the location counter. This technique is particularly useful, for example, when a variant field does not exactly map onto another variant.

File controls

The file control directives described in this section let you create and access files other than the current source text files during assembly. They are the following:

INCLUDE	Insert source text from another file
DUMP	Write the current global symbol table to a file
LOAD	Read a file into the current global symbol table
ERRLOG	Create an error-listing file

These directives are discussed in detail below. An additional file control directive, MACLIB, is reserved for future implementation.

File search rules

The Assembler searches in several directories for files specified in INCLUDE and LOAD directives. If the directive is given a full pathname (a name containing at least one colon but not beginning with a colon), it opens the specified file. If the directive is given a partial pathname (a name that either starts with a colon or contains no colons), it searches the accessible directories for the file in the following order:

1. The current directory
2. The directory that contains the current input file
3. The directory or directories specified by an –i Assembler option, in the order specified. Assembler options are described in Appendix H.
4. The directory or directories specified in the {AIncludes} MPW Shell variable. Shell variables are discussed in *Macintosh Programmer's Workshop Reference.*

The foregoing search rules are implemented by prefixing the specified partial filename with the name of the directory being searched.

INCLUDE: Take source text from another file

[macro-label] INCLUDE *str-expr*

The INCLUDE directive causes the Assembler to accept source input from a specified file. *Str-expr* is a string expression whose value is the name of the file. The file is said to be **included.**

The Assembler takes input from the included file until it reaches the end of the file. It then resumes taking input from the original file, starting with the line following the INCLUDE directive. The only time the Assembler does not switch back to the original file is when it encounters an END directive in the included file.

An included file may itself include another file. Included files may be nested in this way up to five levels deep. When looking for included files the Assembler follows the procedure described above under "File Search Rules."

INCLUDE directives are not permitted in macro definitions.

DUMP and LOAD: Write and read symbol table files

[*macro-label*]	DUMP	*str-expr*
[*macro-label*]	LOAD	*str-expr*

The DUMP and LOAD directives let you store and retrieve the Assembler's global symbol tables in external files. This capability helps speed assembly by letting the Assembler access often-used symbol tables from a file instead of building them repeatedly. Symbol tables stored in an external file are said to be "dumped." When they are retrieved, they are "loaded."

The value of *str-expr* is the name of an external file. With the DUMP directive the Assembler creates a new file, or overwrites an old file, of that name.

With the LOAD directive, the Assembler searches for the specified file according to the rules given above under "File Search Rules."

Symbol tables are created by the Assembler from the source text. They are kept in memory different lengths of time, depending on the scope of the symbols in the table. Local symbol tables for code modules are purged at the end of the assembly of each module. Global symbol tables—containing module identifiers, data module field identifiers, and all symbols defined outside of code modules, including macro definitions—are kept for the duration of the assembly process.

By using DUMP, you can tell the Assembler to write the following items from the current global symbol table to the file named by *str-expr*:

- template definitions (including type information)
- absolute equates
- equates to imported identifiers
- register equates
- OPWORD definitions
- imported identifiers
- macro definitions

DUMP does not write these items:

- macro variables
- data module identifiers
- data module field definitions
- data type information
- code module identifiers
- code module labels
- information about exported identifiers

You use LOAD to read a previously dumped symbol table into current memory. When a dumped symbol table is loaded, the loaded symbols are merged with the current global symbol table. This means that there must not be duplicate definitions; the Assembler will report, as an error, any conflict between a current symbol table entry and a loaded entry. A symbol entry in a loaded table will not override an already existing symbol entry.

You can use the LOAD directive only outside modules and templates.

The Assembler writes files containing dumped symbol tables in a format that is more compact than the original source files used to produce the symbol tables. Hence it is advantageous to use DUMP and LOAD when you are using a large number of equates—for example, with the standard Macintosh equates. If you do, your dumped files will require substantially less disk space, than the corresponding source files. The assembly process will go faster as well, because the Assembler will not need to scan all the external equate source files.

ERRLOG: Specify Error Log File

[macro-label] ERRLOG *str-expr*

The ERRLOG directive lets you create a separate log file containing all the error messages reported by the the Assembler. Its name will be thevalue of *str-expr*. All errors and warnings reported in the listing file will also be copied to this error log file.

At the start of the assembly process there is no preset error log file, unless one is specified by an **-e** option in the Assembler command string. You specify one with the first ERRLOG directive in your source text. However, the file is not actually created until there is an error or warning message to write into it.

You can also switch error log files by specifying another ERRLOG directive with a different filename. You can terminate error logging by using ERRLOG with a null string for its filename.

Listing controls

The listing control directives let you control the layout and content of the listing file that the Assembler produces during the assembly process. They include the following:

PAGESIZE	Specify the listing page size
TITLE	Define a title for the listing header
PRINT	Control miscellaneous listing options
EJECT	Start a new page in the listing
SPACE	Insert blank lines in the listing

These directives are discussed in detail below. In addition, you can control the listing font and font size by using the `-font` Assembler option described in Appendix H.

The assembly listing format is described in Appendix C.

PAGESIZE: Specify listing page size

[*macro-label*] PAGESIZE [*lines*][,*width*]

The PAGESIZE directive lets you specify the number of lines and the number of characters per line that the Assembler sends for each page of the listing. Both the *lines* and *width* parameters must be absolute expressions; they cannot contain any forward, undefined, or imported references.

The *lines* parameter indicates how many text lines the Assembler sends to the listing file between successive form feed characters (ASCII $0C). Its value must be greater than 29. Each page also contains six header lines; therefore the actual page length is *lines* + 6. If you omit PAGESIZE in your source text or include PAGESIZE without a *lines* value, the Assembler will assume a default value of 75 text lines. With the six header lines, this makes a default page length of 81 lines.

The *width* parameter indicates how many characters the Assembler sends to the listing file between successive return characters (ASCII $0D). Its value must lie in the range 70..160. The Assembler uses this number to right-justify the date and page entries in the header and to truncate the source line display in the listing. If you omit PAGESIZE in your source text or include PAGESIZE without a *width* value, the Assembler will assume a default line width of 126 characters. The default values for *lines* and *width* are based on printing listings in seven-point Courier on a LaserWriter™ printer.

TITLE: Specify title line for listing

[*macro-label*] TITLE *str-expr*

The TITLE directive lets you specify a title line to be placed in the header information of the listing file. This title remains in effect until the next TITLE directive. The *str-expr* may contain a maximum of 80 characters; the Assembler will truncate it if it is either longer than 80 characters or too long to fit in the header.

When the Assembler encounters a TITLE directive in your source text, it performs an EJECT action to terminate the current listing page. The new title appears on the next page, with the TITLE directive listed first below the new page's header.

PRINT: Control listing information

[macro-label] PRINT *parameter,...*

The PRINT directive lets you control whether or not the Assembler creates a listing, and if so, what information it contains. It may contain from 1 to 13 parameters, separated by commas, in the operand field. The parameters may appear in any order. The possible parameters for any PRINT directive are shown in Table 4-3 and explained below; those that are underlined in Table 4-3 are preset. Those values are in force unless you specifically override them.

Table 4-3
PRINT directive parameters

Parameter	Action
ON	Send lines to the assembly listing file
OFF	Do not send lines to the assembly listing file
GEN	Show macro expansions
NOGEN	Do not show macro expansions
PAGE	Allow automatic page ejects
NOPAGE	Cancel automatic page ejects
WARN	Show warning messages
NOWARN	Do not show warning messages
MCALL	Show macro call statements
NOMCALL	Do not show macro call statements
OBJ	Show generated object code
NOOBJ	Do not show generated object code
DATA	Show up to 255 bytes of generated data
NODATA	Show only the first line of generated object data
MDIR	Show macro directive lines
NOMDIR	Do not show macro directive lines
HDR	Show header lines
NOHDR	Do not show header lines
LITS	Show generated literals
NOLITS	Do not show generated literals

STAT	Show assembly status
NOSTAT	Do not show assembly status
SYM	Show symbol tables
NOSYM	Do not show symbol tables
PUSH	Save current print status
POP	Retrieve saved print status

ON allows a listing file only if you specified a listing filename when invoking the Assembler. OFF suppresses listing until PRINT ON occurs. PRINT OFF directives are not listed in the listing file, regardless of any other parameters they may contain.

GEN and NOGEN control the listing of macro expansion lines. Macro directives appear only if PRINT MDIR is also in force.

PAGE and NOPAGE control whether or not the Assembler sends automatic form feed characters to the listing file.

WARN and NOWARN control both the display and counting of warning messages.

MCALL and NOMCALL control the listing of macro call statements.

OBJ lets the Assembler list the generated object code or data for each listed line. Generated object code is always shown in full. Up to 18 lines (about 255 characters) of generated object data are shown if PRINT DATA is also in force. If PRINT NODATA is in force, only one line of generated object data is shown. NOOBJ suppresses all listing of object code and data. This results in a briefer listing that shows only source text lines and their addresses. If you include PRINT NOOBJ in your source text it should be the first line, to avoid changes in format after the listing has begun.

DATA and NODATA control whether object data is shown in full (up to 18 lines) or limited to one line. These parameter values are effective only if PRINT OBJ is in force.

MDIR and NOMDIR control whether or not macro directives (including conditional - assembly directives and SETA and SETC directives) are shown in the listing. PRINT GEN,NOMDIR lets you list macro expansions without listing any of the macro control statements that produced them.

HDR and NOHDR control whether or not header lines are printed in the listing. HDR is effective only if PRINT PAGE is in force.

LITS and NOLITS control the listing of literals produced by PEA and LEA machine instructions. If LITS is in force, literals are printed at the end of the code module in which they were produced.

STAT and NOSTAT control showing the assembly status in the listing. If PRINT STAT is in force, each module identifier is listed to the diagnostic output file when the Assembler encounters it in the source text.

SYM and NOSYM control showing symbol tables in the listing, including local symbol tables at the end of each module and the global symbol table at the end of the assembly.

PUSH saves the current PRINT parameter values before processing new ones. POP retrieves the most recently saved values and places them in force. You can save up to five different sets of values in this way. PRINT directives containing PUSH or POP are not listed. A typical use of PUSH and POP is to save the current listing format before listing a macro in a different format, then restore it afterward. By using the &SETTING function described in Chapter 6, you can access the listing format at any level on the stack.

To override the preset values of one or more parameters, write a PRINT directive in your source text containing only the parameter values you want to change. Repeating a parameter value already in force will not cause an assembly error.

❖ *Note:* You can also accomplish most of the PRINT directive actions by using the `-print` Assembler option described in Appendix H. However, PRINT directives in your source text override the `-print` Assembler option.

EJECT: Start new listing page

[*macro-label*] EJECT [*lines*]

EJECT causes the next line of a listing to appear at the top of the next page. The EJECT directive itself is not listed. If EJECT occurs when the next line would already be at the top of the next page, it has no effect.

The parameter *lines* is optional. If present, it must be an absolute expression without any forward, undefined, or imported references, with a positive value greater than zero. *Lines* makes EJECT conditional on whether the specified number of lines is available on the current page. If the lines are available, the Assembler takes no action; if not, it ejects a new page in the listing. This lets you make sure that a section of your source text is listed all on one page.

SPACE: Insert blank line in listing

[*macro-label*] SPACE [*lines*]

SPACE lets you insert one or more blank lines into your listing. The SPACE directive itself is not shown. The optional parameter *lines* indicates how many blank lines to insert (from one to the current page length). It must be an absolute expression without any forward, undefined, or imported references. If the parameter is omitted, the Assembler inserts one blank line. If the value of the *lines* parameter exceeds the number of lines remaining on the page, SPACE has the same effect as EJECT.

Chapter 5

Defining and Using Macros

A **macro** is a previously defined sequence of statements and/or directives that the Assembler processes when it encounters a corresponding **macro call.** During macro processing, the Assembler usually generates new source text. The MPW **macro directives** provide simple but powerful tools with which you can define macros and determine where and how they are processed. Taken together, these tools are called the **macro language,** which is implemented by a portion of the MPW Assembler called the **Macro Processor.**

This chapter and the next two chapters explain how to use the MPW Assembler macro language. Here is a brief summary of what they contain:

- The present chapter tells you how to write **macro definitions** and macro calls. A macro definition is a named section of source text containing the statements and/or directives that constitute a macro. Macro definitions are set off by the directives MACRO and ENDM or MEND. A macro call is a statement that invokes a macro by name, causing the Macro Processor to **expand** the call. When the Macro Processor expands a macro call, it replaces the macro call statement with the contents of the macro definition, substituting actual values for certain of its variables and parameters. Macro expansion makes it easy for you to generate lengthy but repetitious source text sequences, by defining a few macros and then calling them repeatedly.
- Chapter 6 discusses **macro variables** and the functions that operate on them. A macro variable is a variable whose value is assigned by the macro language. The macro language contains a set of functions that let you form expressions out of macro variables, constants, and Assembler system variables. When such expressions occur in the body of a macro, they are replaced by their current values every time the Macro Processor expands a call to that macro. You can use macro variables outside of macros (if they were originally declared to be global) and thereby access their current values outside or inside macros. There are three kinds of macro variables: **symbolic parameters, Assembler system variables,** and **SET variables.** These three types differ in the ways they acquire values. Symbolic parameters acquire values when the Macro Processor expands macro calls; Assembler system variables are assigned values by the Assembler itself; and SET variables are assigned values by explicit macro directives.
- Chapter 7 describes the MPW Assembler macro directives, which are instructions you give to the Macro Processor. Using the macro directives, you can determine whether the Assembler will process or ignore sections of your source text, based on the values of boolean control expressions. This facility is called **conditional assembly.** It is a powerful tool for creating and controlling source text structures.

Macro expansion

Every line of source text is interpreted first by the Macro Processor, which handles it as follows:

- If the source text line does not contain any elements of the macro language, the Macro Processor simply passes it unaltered to the rest of the Assembler for assembly-language interpretation.
- If the line contains any macro variables or functions but is not a macro directive statement, the Macro Processor replaces the expressions with their current values. It then passes the result to the rest of the Assembler for assembly-language interpretation.
- If the line is a macro directive statement, the Macro Processor handles it according to the rules defined in Chapter 7. In this case it interprets and acts upon any macro expressions that the line contains, instead of replacing them with their current values.

Thus macro directive statements are handled entirely by the Macro Processor; they are not passed to the rest of the Assembler. Machine instruction statements and other directive statements are handled by the rest of the Assembler, after the Macro Processor has converted any macro expressions they might contain to actual values.

You should remember certain rules that the Macro Processor follows when preparing statements to be passed to the rest of the Assembler:

- An element to be replaced, such as a macro parameter or a variable reference, may not be continued across a line boundary.
- An element to be replaced may not contain another element (such as an array index or function parameter) that must be replaced first.
- All fields are expanded, including comments. If you want to refer to a macro variable in a comment that is part of a macro call or prototype statement, you must precede it by a **backquote** (`) to force the Macro Processor to treat it literally instead of replacing it with its value. Similarly, the name of a macro function must be preceded by an extra ampersand.

Scope of macro symbols

As explained in Chapter 2 under "Scope of Definitions," the scope of a definition is the range of source text in which the defined identifier can be accessed by code or data statements. A variable is called "accessible" within the scope of its identifier. The **lifetime** of a variable is the duration of the assembly process over which it is accessible. These rules govern the scope of macro symbols and the lifetime of macro variables:

- Variables declared within a macro, including all its parameters, are accessible only in subsequent statements of that macro. Their lifetime is the duration of the current expansion of the macro; they are not accessible in future or nested invocations of the same macro. Local variables with the same identifier that are defined in other macros or outside of macros are different variables.
- Variables declared local outside of any macro definition are accessible only in subsequent statements outside macros. You can think of them as program-level variables. Their lifetime is the remainder of the assembly process after their definition. Local variables of the same identifier defined inside macros are different variables.
- Variables declared global in a macro are accessible in all subsequent statements of the program. Their lifetime is the remainder of the assembly process after their definition.
- When the lifetime of a local variable or the scope of its identifier overlaps that of a global variable, the local variable takes precedence and the global variable becomes temporarily inaccessible.

Defining macros

A macro definition is a sequence of statements that tells the Macro Processor the name of a macro, the format of its call, and the source text to be generated when the macro is called. It consists of four parts:

- The **header** directive (MACRO)
- The **prototype** statement
- The macro **body**
- The **trailer** directive (ENDM or MEND)

Here is the format and syntax of a macro definition:

	MACRO	header
[*label*]	*name*[.*macro-qualifier*] [*parameter-list*]	prototype
	machine instruction and/or directive statements	body
[*macro-label*]	{ ENDM | MEND }	trailer

Here are some rules about macro definitions:

- Every macro must be defined before it can be used; that is, its definition must precede any calls to the macro.
- You can write macro definitions anywhere in a program except within other macro definitions.

- A macro whose definition appears within a conditional section of source text (for example, within a section delimited by the IF directive) will not be defined if the conditional branch causes the Macro Processor to skip over its definition.

The parts of a macro definition are discussed below.

MACRO and ENDM or MEND: Delimit a macro

The MACRO and ENDM directives begin and end the macro definition. In place of ENDM you may write MEND.

MACRO takes no labels or operands. ENDM may be preceded by a macro label, but such a label can be referred to only by a GOTO directive (described in chapter 7).

The prototype statement

The macro prototype statement specifies the name of the macro being defined and the format of calls to the macro. The prototype statement also establishes the identifiers of the macro parameters, if any. The basic form of the macro prototype statement is as follows:

label field	**operation field**	**operand field**
[*label*]	*name*[.*macro-qualifier*]	[*parameter-list*]

For the full syntax of the macro prototype statement, including parameter types, default values, and keywords, see "Parameter Types and Default Values" under "Calling Macros," below.

The only required part of the macro prototype statement is the macro's name, which identifies it for later calls. The macro name must follow the rules for identifiers given in Chapter 2. It may not be the same as the name of any machine instruction, Assembler directive, or other macro used in the same assembly.

In the other parts of the prototype statement—the label, the macro qualifier, and the **parameter list**—you may write only symbolic parameter identifiers. These are valid assembly-language identifiers preceded by an ampersand (&). They are described below under "Symbolic Parameters." For the parameter list, you may write a series of symbolic parameter identifiers separated by commas. The macro qualifier, if present, must follow the macro name with a period as a separator.

The label field in a macro prototype statement is discussed below under "Macro Call Labels."

You may continue a prototype statement on the next source text line at any point after the macro name (or macro qualifier, if present). To continue a line, you must break it after a comma and insert a backslash as a continuation character, as in the following:

```
MyMacro     &parmA,&parmB,&parmC,\
            &parmD,&parmE
```

You can continue a prototype statement indefinitely.

The Macro Body

The macro body consists of the set of machine instruction and/or directive statements between the macro's prototype statement and its trailer. It may contain any or all of three kinds of source lines:

- **model statements,** which the Macro Processor uses as a model to generate actual Assembler statements or directives
- **macro directives,** which control the process of macro expansion but generate no statements
- **inner macro calls,** which invoke other macros

Just like other assembly-language statements, model statements consist of four fields, some of which may be omitted: the label field, the operation field, the operand field, and the comment field.

The label field can be omitted or may contain a symbol or symbolic parameter. A symbolic parameter in a macro label field whose value is an asterisk (*), period-asterisk (.*), or semicolon (;) will not turn a model statement into a comment, since the Macro Processor decides whether or not it is a comment before substituting values for its macro variables.

The operation field may contain any MPW assembly-language instruction or directive, including macro directives, or any variable symbol. It may not, however, contain the an INCLUDE directive. If it contains the words Endm or Mend, the macro will terminate at that point.

The operand field within a model statement follows the same rules as that in other statements and directives. The operand fields of some macro directives, however, require embedded keywords. In such cases the operand field terminates when the required syntax is complete, rather than with the first space or tab.

The comment field within a model statement is discussed below.

Macro Comments

You may add comments to macro prototype statements, subject to these rules:

- If your prototype statement has no parameters, you must begin your comment with a semicolon.
- If a parameter list is present and you want to write a comment, you must separate your comment from the end of the parameter list by at least one space or tab (with `BLANKS ON`) or a semicolon (with `BLANKS OFF`).
- If the parameter list is continued on a second line, you may insert comments on individual lines. Finish each line with a comma and a backslash continuation character. The Macro Processor will treat all text remaining on the line after the continuation character as a comment. Here is an example:

```
MyMacro     &parmA,             \ Comment on ParmA
            &parmB, &ParmC,     \ Comments on ParmB and ParmC
            &parmD                Comment on ParmD
```

You may add comments to macro model statements in the same way as you add them to ordinary statements, but you must follow one additional rule. To refer to a macro variable or function in a comment, you must precede it with an additional ampersand; otherwise the Macro Processor will substitute its value during macro expansion.

You can write a line containing only a comment in a macro if it starts with an asterisk (*) or a semicolon (;). Additionally, you may write a special variety of comment line beginning with period-asterisk (.*), which is not stored in the macro definition. In an Assembler listing, a comment beginning with a period-asterisk will be listed in the macro definition but not in any macro expansion; comment lines that begin with a sole asterisk or a semicolon will be listed both in the definition and in all expansions.

The following example demonstrates all the types of macro comment lines. The line numbers are for reference only.

```
1           MACRO
2           DbgHead                             ; Macro DebugHead
3    .* Puts the Pascal entry code in front of a
4    .* subroutine so MacsBug can identify it
5    * >>> Debug Header <<<
6           LINK        A6,#0                   ; Set up stack frame
7           ENDM
```

The comment on line 2 is preceded by a semicolon because the prototype statement has no parameters. The comments on lines 3 and 4 explain what the macro does. They are intended to appear only in the listing of the definition, so they are preceded by period-asterisks. The comment on line 6 is a note that appears in the expansion of each call, so it is preceded by an asterisk alone. This causes the Macro Processor to list it both in the definition and in all expansions. The comment on line 5 is a model statement comment.

Symbolic parameters

Symbolic parameters are a special type of variable symbol, to which the Macro Processor assigns values when a macro is called. They are used to pass information to the macro from macro call statements.

Symbolic parameters must be defined in the prototype statement. Only then can you refer to them in statements in the macro body. The values of symbolic parameters are assigned when a macro is called and cannot be subsequently changed except by another call to the same macro.

The following macro definition illustrates the use of symbolic parameters. The line numbers are for reference only; they would not appear in an actual source text.

```
1           MACRO
2           Incr        &src,&dest              ; Get src, increment, move to dest
3           MOVE.W      &src,D0                 ; Move &src to temp register
4           ADDQ.W      #1,D0                   ; Increment temp register
5           MOVE.W      D0,&dest                ; Move temp register to &&dest
6           ENDM
```

When the macro Incr is called, the Macro Processor replaces all references to symbolic parameters in the macro body by corresponding values in the macro call (except when the reference is preceded by an additional ampersand). Thus when the Macro Processor encounters the macro call

```
            Incr        Alpha,Beta
```

it replaces it with the following lines:

```
3           MOVE.W      Alpha,D0                ; Move Alpha to temp reg
4           ADDQ.W      #1,D0                   ; Increment temp reg
5           MOVE.W      D0,Beta                 ; Move temp reg to &dest
```

Notice that the `&src` parameter defined on line 2 of the macro definition appears twice on line 3 in the macro body. When the macro is expanded, both references to `&src` are replaced with the parameter value `Alpha`. In line 5, however, the second occurrence of `&dest` is not replaced with the value `Beta`, because it is preceded by a second ampersand. The process of macro expansion is explained in more detail below under "Calling Macros."

Concatenating symbolic parameters

When symbolic parameters occur outside macro directives, and are not part of SET variable subscripts or function arguments, you can concatenate them with other characters or symbols by simply putting the objects to be concatenated next to each other, without any intervening spaces. Potential ambiguities arise when a character, number, left square bracket, or period is concatenated to the right of a symbolic parameter reference, because the Macro Processor interprets these characters as part of the parameter reference. In such cases you must terminate the parameter reference with a period, to distinguish the end of the parameter identifier from the characters concatenated to the right.

When a symbolic parameter or a variable reference is followed by a period, the Macro Processor replaces the symbol and the period with the symbol's value when the macro is expanded. The period does not appear in the generated statement.

The table below shows some sample results from the concatenation of a parameter with various combinations of text, other parameters, and special characters. The column on the left contains examples of concatenation with the parameter `&param`. The column on the right shows the result of macro expansion when the value of `&param` is A:

Expression	Result
`&param.B`	AB
`&param..B`	A.B
`&param.(B)`	A(B)
`B&param`	BA
`B,&param`	B,A
`B2&param`	B2A
`&param.2B`	A2B
`&param,.2B`	A,.2B
`&param&param`	AA
`&param.&param`	AA
`&param..&param`	A.A

The following examples illustrate some variations that often cause problems:

Expression	Result	Notes
`&Param.[1]`	`A[1]`	Period makes it not a sublist reference
`&Param[1]`	null	Sublist reference, but parameter value is not a list, yields null string
`&Param.B`	AB	
`&ParamB`	error	If `&ParamB` is not defined

The first two examples look like attempts to write parameter sublist references, as described below under "Calling Macros"; they also have the same form as subscripted SET variable references, discussed in Chapter 6 under "Set Array Variables." The final example represents the common mistake of forgetting to terminate a parameter identifier with a period when it is followed by other characters, causing the Assembler to read the string as one parameter name, `&paramB`.

Calling macros

A macro call is an instruction to the Macro Processor to insert at that point in the source text the statements and/or directives specified by the macro definition. The inserted source text replaces the macro call statement.

The format and syntax of a macro call is as follows:

label field	**operation field**	**operand field**	**comment field**
[*label*]	*macro-name*[.*macro-qualifier*]	[*parameter-list*]	[*comment*]

The contents of the operation, operand, and comment fields of a macro call are summarized below; the contents of the label field is discussed in the next section.

The operation field contains the name of the macro to be called. The macro being called must have been defined previously in the assembly process.

The operand field contains information that the macro call passes to the body of the macro. Thus the order of operands in a macro call statement must correspond to the order of the symbolic parameters in the prototype statement of the corresponding macro definition. Parameters of this type are called **positional parameters**. The parameters defined in a macro definition are called **formal parameters**, while the corresponding parameters specified in a macro call are called **actual parameters**. The Macro Processor assigns the values of the actual parameters to the formal parameters when it calls a macro.

When BLANKS OFF is in effect, operands specified in a macro call must be separated by commas, with no intervening spaces. The first space not embedded inside a quoted string will terminate the operand field (as well as the parameter list) and begin the comment field. With BLANKS ON in effect, leading and trailing blanks in the operand field are ignored and comments must be preceded by semicolons. The form required for individual operands in both cases is discussed in detail below.

The comment field is ignored by the Macro Processor. The operand and comment fields of macro calls may be continued on more than one line, using the conventions applicable to prototype statements. For details, see "The Prototype Statement," under "Defining Macros," earlier in this chapter.

Macro call labels

The label field of the macro call may contain an identifier. The Macro Processor interprets macro call labels according to these rules:

- If the prototype statement of the macro being called does not contain a label, the macro call label refers to the current location counter value at the point of the macro call. In code modules, this is a reference to the first generated statement of the macro. In data modules, however, the current location counter is not necessarily aligned with the next defined data item.
- If the prototype statement of the macro being called contains a label, the value of the call's label is passed to the macro, as if the prototype's label were an operand. However, the syntax of the label identifier is restricted to the rules covering other identifiers; you cannot declare it as a sublist or keyword, type it, or give it a default value, as you can with a normal operand.
- When the value of the call's label is passed to the macro as just described, it may be used for any purpose—as a label on a statement in the body of the macro, or as a variable.

For example, suppose a call to the macro `incr`, described earlier under "Symbolic Parameters," is contained in the following code fragment:

```
            BRA.S       x1
            - - -
    x1      incr        D2,D3
```

Because the macro definition of incr does not contain a label in its prototype statement, the branch to `x1` will automatically be interpreted as a branch to the first generated statement of `incr` (line 3).

Now suppose the same call is used with a different definition of `incr`:

```
1           MACRO
2   &y1     Incr        &src,&dest              ; get src, increment, move to dest
3           MOVE.W      &src,D0                 ; move &src to D0
4           ADDQ.W      #1,D0                   ; increment D0
5   &y1     MOVE.W      D0,&dest                ; move D0 to &&dest
6           ENDM
```

In this case the branch will go to line 5, because the actual value `x1` is now passed to the formal label `&y1`, which is a label on the third generated line of `incr`.

Operand syntax

The value of any macro call operand may be numeric or may be any sequence of up to 255 characters. In the latter case, you must observe the conventions set forth below.

Paired Single quotes

A macro call operand may contain one or more strings enclosed by single quotes ('), called **quoted strings**. However, quoted strings may themselves contain single quotes. Single quotes that are part of the string are written as two adjacent single quotes. Thus, if the first single quote of a quoted string is numbered 1, then the string ends with the first even-numbered single quote that is not followed immediately by another single quote. The first and last single quotes of a quoted string are called **paired single quotes**.

The following sample operand consists of a sequence of characters that contains two quoted strings:

```
'='10'C''C's
```

The first and second apostrophes are paired, as are the third and last. The Macro Processor interprets the fourth and fifth apostrophes as one embedded apostrophe. Note that this sample operand contains characters in addition to those in the strings.

Paired parentheses and brackets

A macro call operand must be balanced with respect to parentheses and brackets; that is, there must be an equal number of left and right parentheses and the *n*th left parenthesis must appear to the left of the *n*th right parenthesis. The same is true for square brackets.

The simplest case of **paired parentheses** is a left parenthesis followed by a right parenthesis without any intervening parentheses or brackets. Similarly, the simplest case of **paired brackets** is a left bracket followed by a right bracket without any intervening parentheses or brackets. If there is more than a single pair of parentheses or brackets, the Macro Processor associates them by repeatedly recognizing and removing such simple pairs.

This method is used because paired parentheses and brackets normally enclose lists, which may be nested. Searching for the simplest pairs lets the Macro Processor recognize the list structure, as well as determine whether any given comma is an interior part of a list or the end of an operand in the parameter list.

The following example contains three sets of paired parentheses and one set of paired brackets:

```
(READ,(src.text,EXT))inBufr(input[2,512])
```

The first and fourth parentheses are paired, as are the second and third and the fifth and sixth. The two brackets are paired.

The Macro Processor ignores any parentheses or square brackets appearing between paired quotes when associating paired parentheses, as in the following example:

```
(N''('closed)
```

Ampersands

To write a literal ampersand in source text, you must write two consecutive ampersands. The Macro Processor interprets any single ampersand as the beginning of a reference to a symbolic parameter, a SET variable or function call, or a Macro Processor system variable. In nested macro calls in which a symbolic parameter is concatenated to one or more preceding ampersands, the Macro Processor interprets the concatenation as a sequence of literal ampersands in which the final odd-numbered ampersand is the beginning of a variable reference.

Commas

A comma delimits the end of every macro call operand unless the comma appears between paired quotes, parentheses, or brackets. For example, the following is a single operand even though it contains several commas:

```
(D0,D1,D2)delim',''right
```

Blanks (spaces and tabs)

The effect of spaces and tabs in a macro call operand depends on the setting of the BLANKS Assembler directive. If the current setting is BLANKS OFF, then any space or tab terminates the operand field, unless the line is continued. The only exceptions are spaces or tabs written between paired quotes, as in the following example:

```
'Value too large'(&Count,10)
```

If the current setting is BLANKS ON, the Macro Processor will retain spaces and tabs if they are followed by more valid operand characters. It will ignore trailing spaces and tabs if they occur at the end of a line or before the comma delimiting the next operand.

Backquotes

The backquote (`) is used in macro prototype and call operands to indicate that the next character should be passed through the Macro Processor without interpretation. Thus an ampersand may be put into a call operand without the Macro Processor interpreting it as the beginning of an & reference if it is preceded by a backquote. For example:

A backquotes can be used to literalize any character in a macro prototype or call operand, including another backquote.

@-labels

When interpreting macro call operands, the Macro Processor assumes that any string beginning with @ is an @-label, unless it is enclosed in apostrophes or preceded by a backquote. For example, when processing the sublist `(A,@2,B)` the Macro Processor will treat the element @2 as an @-label. Implementation restrictions force it to make this assumption in order to encode the scope of the caller's @-label inside the macro. If you want to begin a call operand that is not an @-label with @, you must literalize it with a backquote.

Omitted or Extra Operands

If you omit an actual parameter from a macro call, you must still include the comma that would have separated the omitted actual parameter from the next parameter. This preserves the positional correspondence of parameters. The Macro Processor gives each omitted actual parameter a value of 0 if the corresponding formal parameter has integer type, or assigns it the null string if the corresponding formal parameter is type string. Parameter types are discussed under "SET Variables" in Chapter 6.

If you omit one or more of the last operands from a macro call, you may also omit the commas that would have separated these final operands. With macros that use the number of actual parameters internally, the Assembler counts trailing commas when determining the number.

These rules are demonstrated in the following sample macro prototype statement and its subsequent macro call:

prototype: `ExampleMac &parm1,&parm2,&parm3,&parm4,&parm5,&parm6`

call: `ExampleMac D0,*+6,,'syntax error'`

In this macro call, the third parameter has been omitted (nothing is between the second and third commas), as have the fifth and sixth parameters; no final commas are required.

To avoid having the Macro Processor assign the null string as the value of an actual parameter omitted from a macro call, you may write the prototype statement so that it assigns default values to omitted parameters. The section "Parameter Types and Default Values," below, tells you how to do this.

It is permissible in a macro call to specify more actual parameters than there are formal parameters defined in the corresponding macro definition. There will, of course, be no formal parameter identifiers by which you can refer to these extra operand values; however, you can still access them by using `&SYSLIST`, as described in Chapter 6 under "Assembler System Variables."

Operand sublists

You can structure a macro call operand as a **sublist.** This lets you refer to a collection of operands in the same way as you would refer to a single operand, while still being able to refer to individual members of the collection. Each operand in a sublist is called an **element**. Sublists are always string types.

A sublist may contain one or more operands, separated by commas and enclosed in paired parentheses. The Macro Processor treats the entire sublist, along with the paired parentheses, as a single operand (limited to 255 characters). Any element of a sublist may itself be a sublist. Some examples are shown here:

```
(A)
(A,B,C)
(A,(B,C),D)
(((A)),B,C)
```

A sublist may be continued on subsequent lines and may contain comments, following the same conventions as macro prototype statements. However, it may not contain more than 255 characters (excluding comments). These conventions are described under "The Prototype Statement", above.

Accessing sublist elements

You can refer to an element of a macro sublist operand in the body of the macro. To do this, you write a sublist as an absolute arithmetic expression enclosed in square brackets immediately after the identifier of the sublist's symbolic parameter. The expression in square brackets represents the **index** (position) of the desired element, with the first element being 1. For example, if `&abc` is the identifier of a sublist parameter, then `&abc`[*n*] refers to the *n*th element in the sublist. The index value *n* must be greater than zero.

If the *n*th element of the sublist is omitted, or if there are fewer than *n* elements in the sublist, then the value of `&abc[n]` will be the null string. If the *n*th element of the sublist is itself a sublist, then `&abc`[*n*,*m*] refers to the *m*th element in the sublist which is the *n*th element of `&abc`. You can create sublist references up to any depth by specifying as many subscripts as necessary, separated by commas, between the brackets. You can use the `&NBR` function, described in Chapter 6, to find out how many elements a sublist has.

If you write a sublist reference to a parameter that is not a sublist, its value will always be the null string.

Note that the left bracket that begins the index expression must follow the parameter identifier without any intervening characters. You should not place a period between the parameter identifier and the left bracket unless you want to concatenate the value of the entire operand with a left bracket.

The following examples illustrate various references to sublist elements. They are based on a parameter `&P` with the sublist `(A, (B,C),D)` as its value. The left column shows sublist element references; the right column shows their corresponding values.

Reference	Value
`&p`	`(A,(B,C),D)`
`&p[1]`	`A`
`&p[2]`	`(B,C)`
`&p[3]`	`D`
`&p[4]`	`null`
`&p[2,1]`	`B`
`&p[2,2]`	`C`

Parameter types and default values

The Macro Processor assumes that all macro parameters are strings. Sometimes, however, it is necessary to declare a parameter as an integer type. You can assign parameter a type by following the parameter identifier in the prototype statement with a colon (:) followed by `STR` or `C` for a character string parameter and `INT` or `A` for an integer (arithmetic) parameter.

If you omit an actual parameter of type integer when calling a macro, the Macro Processor gives it a value of 0. If you omit a string parameter, the Macro Processor assigns it the null string.

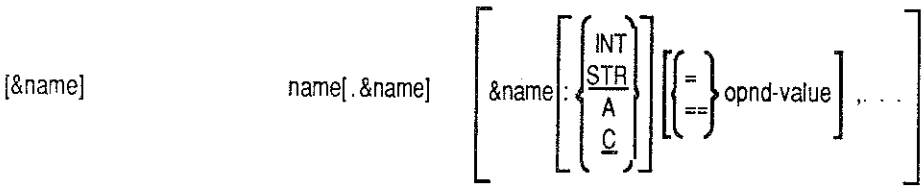
The actual value passed to a parameter of integer type must be an integer constant, because the Macro Processor interprets it using the `&STRTOINT` function. Values of integer expressions must be passed as strings and then converted inside the macro, using the `&EVAL` function. `&STRTOINT` and `&EVAL` are described in Chapter 6.

You can assign default values to macro parameters by following their identifiers (or type specifications) in the prototype statement with an equal sign (=) followed by the default value. Each such parameter will take on the default value if it is omitted in a macro call.

The following example of a macro prototype statement defines two parameters of different types with different default values; `&Sz` is an integer with a default value of 4, while `&reg` is a string with a default value of `'D0'`.

```
MACRO
BumpSz    &Sz:Int=4, &reg=D0
- - -
ENDM
```

The full syntax of any macro prototype statement is therefore the following:



The double equal sign (==) preceding the default operand value *opnd-value* in this syntax diagram identifies keyword parameters. Keyword parameters are discussed later in of this chapter, under "Keyword Macros."

Nesting Macros

Macros may be called from the bodies of other macro definitions. Such macro calls are termed **inner macro calls,** because they are executed within a macro definition. They are also termed "nested calls." A macro call that is not within the body of a macro definition is an **outer macro call.**

When the Macro Processor encounters an inner macro call during macro expansion, it suspends processing the current macro and expands the inner macro. When it has finished expanding the inner macro, it resumes expansion of the outer macro. The macros invoked by inner macro calls may call other macros, so there may be any number of macros suspended in mid-process while the current macro is being expanded. The sequence of macros suspended in this way is called the current **macro call chain.**

The definition of a macro called by an outer macro call may contain any number of inner macro calls. The outer call is termed the **first-level call;** all inner calls in a first-level macro are termed **second-level calls;** calls within second-level macros are third-level calls; and so on. The number of each such **dynamic nesting level** is available in the Assembler listing if macro expansions are being listed.

Nesting levels refer only to the way the Macro Processor actually expands macros in a given assembly. Any particular inner macro call may have different nesting levels at different points in an assembly, because the macro containing it may be called from a variety of places, along a variety of call chains. The number of levels of nested macros that your program can call depends only upon the complexity of your macro constructs and the memory available.

The following example demonstrates how a macro may be called from within another macro. This is the definition of `Incr`, the macro to be called:

```
            MACRO
            Incr        &src,&dest
            MOVE.W      &src,D0
            ADDQ.W      #1,D0
            MOVE.W      D0,&dest
            ENDM
```

Now here is the definition of a macro, Incr2, that calls Incr:

```
            MACRO
            Incr2       &a,&b,&c
.* increment &a and &b, put sum in &c
            Incr        &a,&a
            MOVE.W      &a,D1
            Incr        &b,&b
            ADD.W       &b,D1
            MOVE.W      D1,&c
            ENDM
```

If Incr2 is called with X, Y, and Z as its three parameter values, the Macro Processor will generate the statements shown below by macro expansion. The line numbers on the left are included only for reference to the example.

```
            Incr2       X,Y,Z                   ; macro call
1           MOVE.W      X,D0                    ; macro expansion
2           ADDQ.W      #1,D0
3           MOVE.W      D0,X
4           MOVE.W      X,D1
5           MOVE.W      Y,D0
6           ADDQ.W      #1,D0
7           MOVE.W      D0,Y
8           ADD.W       Y,D1
9           MOVE.W      D1,Z
```

Notice that lines 1 through 3 are generated by the first inner macro call to `Incr`. Lines 5 through 7 are generated by the second inner macro call. The other lines are generated from the `Incr2` macro.

Keyword macros

Keyword macros provide an alternate way to pass parameter values during macro expansion. In a keyword macro definition, you can specify parameters in any order and give them default values. The Macro Processor identifies different parameters by their keywords rather than by their position in the parameter list. As a result, when you call a keyword macro, you need specify only those parameters that require values different from their default values.

The different ways you can write macros are distinguished by this terminology:

- **Positional macros** are the kind discussed previously in this chapter. Their parameters are distinguished by their positions in the macro prototype statement.
- **Keyword macros** are the kind described below.
- **Mixed-mode macros** contain both positional and keyword parameters. They are described at the end of this chapter.

❖ *Note:* You cannot use `&SYSLIST` to access keyword parameters.

Defining keyword macros

The prototype statement that defines a keyword macro is like the prototype statement for a positional macro (described at the beginning of this chapter under "Defining Macros"), except that each parameter identifier in the parameter list is followed by two equal signs (==). This tells the Macro Processor that the parameter identifiers are **keywords.**

The two equal signs may optionally be followed by an integer constant or string giving the parameter's default value. This constant must have the same type as the parameter.

The following are examples of valid keyword prototype operands:

```
&recSize==12
&recSize:INT==12
&inLine==
&err==(12,'syntax error')
```

The following are examples of invalid keyword prototype operands:

`InLine`	No & (not a parameter)
`&recSize`	No equal signs
`&blkSize ==512`	Space before ==
`&x=2`	Single equal sign; interpreted as a positional operand with a default value

The following is an example of a keyword prototype statement with a symbolic parameter in the label field and three keyword parameters in the operand field:

```
&lab    CopyBuff      &src==InBuff,&dest==,&count==512
```

Note that the second parameter does not have a default value.

Calling keyword macros

Once you have defined a keyword macro, you can tell the Macro Processor to expand it and insert it into your source text with a **keyword macro call** directive of the following form:

[*label*] *macro-name* [*keyword*=[*value*]] ,...

The keyword operands are keywords without their & prefixes, each of which is followed immediately by an equal sign (=). Each equal sign may optionally be followed by a value. Such operand values must conform to the same rules as operand values in positional macro calls (see "Calling Macros," earlier in this chapter).

The keywords specified in the macro call must have the same identifiers as the keyword parameters defined in the keyword macro definition (without initial ampersands). The Macro Processor does not distinguish between uppercase and lowercase in keywords.

The following are examples of valid keyword macro call operands:

```
RecSize=1024
dest=printBufr
Count=
```

The following are examples of invalid keyword macro call operands:

`&Err=(8,'bad input')`	Starts with &
`sysIn =foo.text`	Space before =
`dest`	No equal sign
`=1024`	No keyword

Here are some rules about writing keyword macro calls:

- You can write keyword operands in any order.
- You need specify only those operands whose values must be different from the default values specified in the macro definition.
- You need not write extra commas for omitted operands.

When the Macro Processor expands a keyword macro, it follows these rules:

- It processes identifiers in the label and operation fields in the same way that it processes such identifiers in positional macro calls.
- In the operand field, it replaces all parameters mentioned in the call with their specified values.
- It replaces all parameters not mentioned in the call with their default values, as specified by the keyword macro prototype statement.
- If a parameter not mentioned in the call has no default value, the Macro Processor replaces it with the null string (type string) or 0 (type integer).
- If a parameter is mentioned in the call but is not followed by a value, the Macro Processor replaces it with the null string (type string) or 0 (type integer).

The following example illustrates the use of keyword macros. It is the same as the example given earlier in this chapter under "Symbolic Parameters" but rewritten with keywords instead of positional parameters. Using keywords improves the macro in four ways: it lets you specify a default temporary register, it gives you the option of leaving the result in the temporary register by simply not specifying a destination register when you call the macro, makes it easy for you to specify an increment value other than 1, and makes the macro call easier to understand.

```
            MACRO
&lbl        Incr        &src==,&dest==,&DReg==D0,&inc==1
&lbl        MOVE.W      &src,&DReg
            ADD.W       #&inc,&DReg
            IF          '&dest'<>'' THEN
            MOVE.W      &DReg,&dest
            ENDIF
            ENDM
```

The following are examples of possible calls to this macro:

```
            Incr        src=D2,dest=D1
            Incr        src=D3,inc=4
            Incr        src=D0,dreg=D1
```

Mixed-mode macros

Mixed-mode macros are macros that contain a combination of positional and keyword parameters.

The prototype statement of a mixed-mode macro resembles that of a positional macro except for its parameter list. In a mixed-mode macro parameter list, you must write all the keyword parameters after all the positional parameters. A positional parameter may not follow a keyword parameter.

The operand lists of mixed-mode macro call directives may include zero or more positional operands, plus zero or more keyword operands. All the actual positional parameters must precede the first actual keyword parameter.

Remember these points when using mixed-mode macros:

- You treat positional operands as if they were in a positional macro and keyword operands as if they were in a keyword macro.
- If you omit a positional parameter, you must insert a comma to indicate the missing position; however, you may omit all trailing commas.
- You can nest all three kinds of macros—positional, keyword, and mixed-mode—in macros of the other kinds. In other words, a macro of any kind may be called as an inner macro from a macro of the same or any other kind.

- The Assembler system variable `&SYSLIST` lists only the positional parameter values in a mixed-mode macro call.

The example given above under "Calling Keyword Macros" could be changed to a mixed-mode macro by removing the equal signs after the `&src` and `&dest` parameters in the prototype statement. Then the following keyword macro call, given earlier:

```
Incr        src=A3,inc=4
```

could be replaced with this mixed-mode macro call:

```
Incr        A3,inc=4
```

In this example, writing a mixed-mode macro definition lets you specify the frequently used `&src` and `&dest` parameters more conveniently in your macro calls, without having to write out their keywords.

Chapter 6

Macro Variables and Functions

Macro variables are variables of the macro language. You can combine them with the functions described in this chapter and use them to control conditional assembly plus certain features of macro expansion. They include the following:

- symbolic parameters, which acquire values when the Macro Processor expands macro calls
- SET variables, which are assigned values by explicit macro directives
- assembler system variables, which are assigned values by the Assembler itself

Symbolic parameters were discussed in Chapter 5. SET variables and Assembler system variables are discussed in this chapter.

SET variables

SET variables help you program the ways that the Macro Processor converts your macro structures into the source text that the rest of the Assembler assembles.

SET variables can have either integer or string values. The MPW Assembler Macro Processor recognizes two different groups of directives, one to handle integer SET variables and the other to handle string SET variables. Similarly, it interprets two classes of expressions formed from these variables. It treats the Boolean expressions used to control conditional assembly directives as a restricted class of integer expression.

You use separate directives to define SET variables and to assign them values. The four **variable definition directives** are these:

LCLA	Defines integer SET variables of local scope (LoCaL Arithmetic)
GBLA	Defines integer SET variables of global scope (GloBaL Arithmetic)
LCLC	Defines string SET variables of local scope (LoCaL Character)
GBLC	Defines string SET variables of global scope (GloBaL Character)

The two directives that assign values to SET variables are these:

SETA	Assigns a value to an integer set variable
SETC	Assigns a value to a string set variable

The MPW Assembler macro language lets you form complex expressions out of macro variables, using a variety of **built-in functions.** The functions that return integer values are these:

&ABS	Returns an absolute value
&ISINT	Tests whether a string contains an integer
&LEN	Returns the length of a string
&LEX	Scans the tokens in a string
&LIST	Divides a string into a list and places the list elements in an array
&MAX	Returns the largest value in a list of integer expressions
&MIN	Returns the smallest value in a list of integer expressions
&NBR	Returns the number of elements in a sublist or parameter list
&ORD	Returns the integer value of a relocatable numeric expression or one-character string expression
&POS	Finds the position of a substring in a string
&SCANEQ	Finds a character in a string
&SCANNE	Finds the first non occurrence of a character in a string
&STRTOINT	Converts a string expression to its integer value

Among the integer functions, the following let you create and manipulate symbol tables:

&NEWSYMTBL	Creates a new symbol table and returns its id number
&ENTERSYM	Enters or updates a symbol in a symbol table
&FINDSYM	Finds a symbol in a symbol table
&DELSYMTBL	Deletes a symbol table

The functions that return string values are these:

&CHR	Converts an integer value to a one-character string containing its ASCII equivalent
&CONCAT	Concatenates string expressions
&DEFAULT	Returns a default string value if a test string is null
&GETENV	Returns the value of an MPW Shell variable
&INTTOSTR	Converts an integer expression to its string value
&LOWCASE	Converts a string to all lowercase
&SETTING	Returns the current setting value for certain other directives
&SUBSTR	Returns a substring of a string
&TRIM	Removes leading and/or trailing spaces and tabs from a string
&TYPE	Returns the type of a symbol
&UPCASE	Converts a string to all uppercase

One function returns either an integer or string value but is ordinarily used as an integer function:

&EVAL	Converts a string expression to either an integer or a string, by evaluating the expression contained in its string argument

The Macro Processor treats SET variables differently, depending on where they occur in the source text. It follows these rules:

- When a SET variable appears in the label, operation, or operand field of a model statement in a macro definition, the Macro Processor replaces the SET variable with its current value regardless of its context. In this situation, SET variables act like symbolic parameters.
- When a SET variable appears inside a macro definition as a label or operand of a macro directive statement, it is replaced only if explicitly allowed by the syntax of the directive or expression where it occurs. In this situation, SET variables act like normal variables.
- SET variables may also appear outside macro definitions, but only in SET variable definitions, SETA or SETC directives, absolute expressions, immediate effective addresses, and conditional assembly directives. If they appear in other kinds of expressions, the Assembler does not recognize them as SET variables.

SET variables and symbolic parameters

SET variables have broader capabilities than the symbolic parameters discussed in Chapter 5 under "Defining Macros." These are their main differences:

- Flexibility of values. Symbolic parameters are assigned values in macro call statements, and these values remain fixed during macro expansion. SET variables acquire values when you use either the SETA or SETC directive. Symbolic parameters are not referred to as variables because their values cannot be changed during macro expansion. SET variable values, on the other hand, may vary during macro expansion as the result of any number of SETA or SETC directives.
- Use outside macros. Symbolic parameters can be accessed only inside macro definitions. SET variables can be accessed both inside and outside of macros. SET variables outside macros are normally used to control conditional assembly.
- Different types. Unless you type them to the contrary in their prototype statement, symbolic parameters are always strings. SET variables, on the other hand, are divided explicitly into string and integer types. Each type has its own functions for creating expressions.
- Global or local scope. Symbolic parameters always have local scope; they may be accessed only in the macro in which they are defined. You can define SET variables so that their scope is local to a macro. However, you can also define SET variables so that their scope is local to your source text outside any macro. Finally, you can define them globally, so that the same SET variable may be accessed in several macros or both inside and outside macros.

LCLA, LCLC, GBLA, and GBLC: Define SET variables

You must define any SET variable before you can give it a value or refer to it in an expression. You use the LCLA, LCLC, GBLA, and GBLC directives to define SET variables. GBLA and GBLC define global integer and string variables, respectively, and LCLA and LCLC define local integer and string variables.

Each of these directives takes a list of one or more SET variable identifiers (*set-var-name* in the syntax diagram above), separated by commas, in its operand field. Every SET variable identifier consists of an ampersand (&) followed by a valid identifier, just like a symbolic parameter identifier. If the SET variable is an array its identifier is followed by a dimension, as described later in this chapter under "Defining SET Array Variables." The Macro Processor ignores uppercase and lowercase distinctions in these identifiers, regardless of the setting of the CASE Assembler directive. LCLA, LCLC, GBLA, and GBLC directives may appear anywhere in your source text, including in macro definitions.

These are examples of valid SET variable definitions:

```
LCLA        &n
LCLC        &args
GBLC        &mainModName,&mainSegName
```

You decide whether to use integer-type or string-type SET variables by analyzing the values the variable must store and how the variable is to be used. Integer variables contain 32-bit values in the range -2147483648..2147483647. String variables contain strings of length 0..255. Such strings may contain any ASCII characters, with these exceptions:

- String constants may not contain the return character (ASCII code $0D).
- String constants may not contain the NUL or SOH characters (ASCII codes $00 and $01) when appearing in macro definitions, because these characters have special significance during macro definition storage.

The choice of whether to use variables of local or global scope is determined by where a variable needs to be accessed. Local scope is sufficient for most purposes, unless a variable must be shared by several macros or preserved between macro calls. The applicable rules are given in Chapter 5 under "Scope of Macro Symbols."

Remember these rules when defining SET variables:

- You cannot use the same identifier for both a symbolic parameter and a SET variable in the same macro. For example, consider the following macro prototype statement:

```
&label      MyMove      &from,&to
```

The Assembler would report an error if you tried to define a SET variable named `&label`, `&to`, or `&from` within the macro—that is, in the same scope.

- You cannot define two SET variables with the same identifier but with different types (integer and string) and with the same or overlapping scopes.
- A global variable defined by a GBLA or GBLC directive must be redeclared whenever it is used in a particular macro.

The Macro Processor assigns **initial values** to SET variables when they are defined, following these rules:

- Integer types are set to 0.
- String types are assigned the null string (length zero).
- Local variables are given initial values whenever they are defined—that is, whenever a LCLA or LCLC directive occurs.
- Global variables are given initial values only the first time they are defined—that is, when the first GBLA or GBLC statement defining the variable occurs.

SETA and integer expressions

Every SET variable of integer type has a 32-bit value. The Macro Processor sets this value to 0 at the first definition of a global integer variable and at every definition of a local integer variable. You can subsequently change the value, using the SETA directive. It has this form:

set-var-name SETA *arith-expr*

The label *set-var-name* must be the identifier of an integer SET variable defined by LCLA or GBLA that includes the SETA directive in its current scope. The operand *arith-expr* must be an absolute integer expression. It may contain any of the following functions, described in this section:

&ABS	Returns an absolute value
&DELSYMTBL	Deletes a symbol table
&ENTERSYM	Enters or updates a symbol in a symbol table
&EVAL	Converts a string expression to either an integer or a string, depending on its actual value
&FINDSYM	Finds a symbol in a symbol table
&ISINT	Tests whether a string expresses an integer
&LEN	Returns the length of a string
&LEX	Scans the tokens in a string
&LIST	Returns number of elements in a list created from a string
&MAX	Returns the largest value in a list of integer expressions
&MIN	Returns the smallest value in a list of integer expressions

&NEWSYMTBL	Creates a new symbol table and returns its id number
&NBR	Returns the number of elements in a sublist or parameter list
&ORD	Returns the integer value of an integer expression or one-character string expression
&POS	Finds the position of a substring in a string
&SCANEQ	Finds a character in a string
&SCANNE	Finds the first non-occurrence of a character in a string
&STRTOINT	Converts a string expression to its integer value

Integer expressions used with SETA must conform to the syntax rules for integer expressions set forth in Chapter 2 under "Expressions." They are also subject to these additional rules:

- As absolute expressions, they must not contain any forward, undefined, or imported references.
- They may contain references to integer SET variables and to symbolic parameters and nonmacro variables and constants with integer values, as long as their scopes include the SETA directive.
- They may refer to untyped symbolic parameters with string values, as long as the string values express valid integer constants. Symbolic parameters of explicit string type must be converted to integer type, using the &STRTOINT function described below.

These are examples of valid integer expressions:

```
33
lab+12
tableBase+(2*&entryNum)
```

&ABS: Return absolute value

```
&ABS(arith-expr)
```

The &ABS function returns the absolute value of the integer expression *arith-expr*. Hence if *arith-expr* is negative, &ABS returns its positive value. The Macro Processor does not check for value overflow.

&EVAL: Evaluate contents of string

```
&EVAL(str-expr)
```

The &EVAL function takes a string expression argument and evaluates the contents of the string as if it were untyped. It returns either a string value or an integer value. It follows the same parsing rules as the Assembler itself; for details, see "Expressions" in Chapter 2. Here are some examples:

Function	Value returned
`&EVAL('2+3')`	integer 5
`&EVAL('&foo')`	`'&bar'` if `&foo` has the value `'&bar'`
`&EVAL(&foo)`	`'fubar'` if `&foo` has the value `'&bar'` and &bar has the value `'fubar'`
`&EVAL('2+x')`	integer 5 if x is equated to 3 (by EQU or SET)

The &EVAL function can be used to evaluate a macro directive parameter whose value is an expression of unknown type. For example, if the formal parameter &a is passed the actual value B+2, the expression &EVAL(&a) returns the value 4 if B=2, rather than the string 'B+2'.

&ISINT: Test string for integer content

```
&ISINT(str-expr)
```

The `&ISINT` function examines the string expression *str-expr* and returns an integer value of 1 if the string expresses an integer, 0 if it does not. The rules by which `&ISINT` recognizes an integer in a string are these:

- It ignores leading and trailing spaces and tabs.
- For nonblank characters, it accepts only numerals and leading plus or minus signs.
- The integer value must not exceed 32 bits.

&LEN: Measure string length

```
&LEN(str-expr)
```

The `&LEN` function examines the string expression *str-expr* and returns an integer value equal to the number of characters in its string value.

&LEX: Parse string lexically

```
&LEX(str-expr, start)
```

The `&LEX` function parses a string into **tokens**—characters or substrings that the Assembler treats as syntactical units—using the Assembler's own lexical scanner. The value of *str-expr* is the string being parsed and *start* is an integer expression whose value is the scan's starting position (the first character of *str-expr* having position 1). `&LEX` returns the integer value of the next token's position, or the current token position if it is the last token of the string.

Each call to `&LEX` sets the value of two Assembler system variables, as follows:

- It sets the value of &SYSTOKEN to an integer value that identifies the kind of token just read.
- It sets the value of &SYSTOKSTR to the actual characters in the token.

The possible values of &SYSTOKEN and &SYSTOKSTR are shown in Table 6-1. These variables are further discussed below under "Assembler System Variables."

Table 6-1
Values returned by &LEX

&SYSTOKEN	&SYSTOKSTR	Meaning
0	identifier	Assembler identifier
1	integer	Assembler integer constant
2	"floating-point constant"	characters enclosed in double quotes
3	'string'	Assembler string constant enclosed in single quotes
4	+	plus (Assembler add symbol)
5	-	minus (Assembler subtract symbol)
6	*	asterisk (Assembler multiply symbol)
7	/ ÷	slash or divide symbol (Assembler divide symbol)
8	//	Assembler MOD symbol
9	++ |	Assembler OR symbol
10	--	Assembler XOR symbol
11	**	Assembler AND symbol
12	=	Assembler equal symbol
13	<> ≠	Assembler not equal symbol
14	<	Assembler less than symbol
15	>	Assembler greater than symbol
16	>= ≥	Assembler greater than or equal to symbol
17	<= ≤	Assembler less than or equal to symbol
18	>>	Assembler shift right symbol
19	<<	Assembler shift left symbol
20	¬	Assembler NOT symbol
21	~	Assembler one's complement symbol
22	(	left parenthesis
23	)	right parenthesis

```
24  [                               left bracket
25  ]                               right bracket
26  {                               left brace
27  }                               right brace
28  ,                               comma
29  .                               period
30  ;                               semicolon and end-of-string symbol
31  :                               colon
32  #                               number sign
33  \                               backslash (Assembler continuation character)
34                                  not used
35                                  all other characters
```

By calling &LEX repeatedly, using the value returned by the previous call for each call's *start*, you can parse a string into its tokens. An &SYSTOKEN value of 30 indicates the last token in the string; &LEX sets &SYSTOKEN to 30 under these conditions:

- if it finds a semicolon
- if the value of *start* is less than 1 or greater than the number of tokens in *str-exp*
- if the current setting of the BLANKS directive is OFF and &LEX finds a space or a tab

If the current setting of BLANKS is ON, &LEX ignores spaces and tabs.

Warning:

The action of &LEX depends on the action of the Assembler's lexical scanner. This could change in future versions of the Assembler.

&LIST: Divide string into a list

```
&LIST(str-expr, str-arr [,delimiter] )
```

The &LIST function divides a string containing a list of items into an array of strings containing the individual items and returns the number of elements in the list. The value of *str-expr* is the string to be treated as a list. The value of *str-arr* is a string containing the name of a SETC array variable represented as a string expression. &LIST sets its array elements with the individual elements contained in *str-expr*. If the array is too small to hold all the elements in *str-expr*, &LIST returns the value of the maximum dimension of the array with a negative sign, to indicate that not all elements could be placed in the array.

&LIST treats *str-expr* as a sequence of substrings, all separated by a unique delimiter not enclosed in paired parentheses, brackets, or single quotes. By default, &LIST assumes the delimiter is a comma, but you can override this default by specifying an optional third argument to &LIST. This argument, *delimiter*, is a string expression that evaluates to a single character which &LIST is to use as the list delimiter.

Here are some examples of values returned by &LIST, assuming that &a is a SETC array variable with a maximum dimension of 4:

Function	Value returned	&a[1]	&a[2]	&a[3]	&a[4]
`&LIST('a,b,c', '&a')`	3	a	b	c	
`&LIST('a/b/c/d', '&a', '/')`	4	a	b	c	d
`&LIST('a,,,d', '&a')`	4	a			d
`&LIST('(a,b)', '&a')`	1	(a,b)			
`&LIST('a(b),([',']))', '&a')`	2	a(b)	([','])		
`&LIST('a,b,c,d,e,f', '&a')`	-4	a	b	c	d
`&LIST('a=b', '&a', '=')`	2	a	b		
`&LIST('(a)):b', '&a', ')')`	2	(a)	:b		

The last two examples show how &LIST can be used to split a string into two parts separated by a delimiter . In the seventh example, the delimiter is the equal sign. In the last example, a right parenthesis is the delimiter. This example illustrates how &LIST looks for delimiters only when they are not enclosed between paired parentheses, brackets, or single quotes. The second right parentheses is not enclosed between any of these characters and is therefore valid as a possible list delimiter.

&MAX: Find maximum in integer list

```
&MAX(arith-expr, ...)
```

The &MAX function accepts a series of integer expressions, separated by commas, and returns the highest value in the list.

&MIN: Find minimum in integer list

```
&MIN(arith-expr, ...)
```

The &MIN function accepts a series of integer expressions, separated by commas, and returns the lowest value in the list.

&NBR: Count sublist elements

```
&NBR( {symb-param | &SYSLIST} )
```

You can use the &NBR function to find out how many elements a sublist contains. The integer value returned by &NBR is equal to 1 plus the number of commas that separate operands in the sublist *symb-param*. When using the value returned by &NBR, keep these rules in mind:

- The empty sublist—with no elements—returns a value of 0.
- If some of the sublist elements are themselves sublists, they may contain commas. &NBR will not count these suborder commas.
- If the argument given to &NBR is not a sublist, &NBR will return 0.
- If the argument given to &NBR corresponds to an omitted operand, &NBR will return 0.

The predefined variable &SYSLIST, which contains the actual parameter list of the current macro, may also be used as an argument to &NBR. With &SYSLIST, &NBR returns the number of positional parameters specified in the call statement that called the current macro. It does not count keyword parameters.

You can use &NBR with arguments that include sublist index expressions—in other words, with sublists that are elements of sublists. In those cases, &NBR returns the number of elements making up the specified sublist. For example, `&NBR(&Parm[3])` returns the number of elements in the sublist `&Parm[3]`. This also works with `&NBR(&SYSLIST[i])`.

&ORD: Return integer value

&ORD(*expr*)

The &ORD function returns the integer value of the expression *expr*. If *expr* has a string value, it may not be more than one character long; &ORD returns the ASCII code of this character. If *expr* is a relocatable numeric expression, &ORD returns its absolute integer value. For example, &ORD(*), written in a code module, lets you use the value of the location counter as an absolute value.

&POS: Find position of substring in string

&POS(*str-expr*$_1$,*str-expr*$_2$)

The &POS function returns an integer value equal to the position of the first occurrence of a substring within a string. The value of *str-expr*$_1$ is the substring; the value of *str-expr*$_2$ is the string. If the substring does not appear in the string, &POS returns a value of 0. Otherwise, it returns the position of the first character of the substring, counting the first character of the string as 1. Here are some examples:

Expression	Result
`&POS('bar','foobar')`	4
`&POS('fu','foobar')`	0
`&POS('o','foobar')`	2

&SCANEQ and &SCANNE: Scan string

&SCANEQ(*ch,str-expr,start*)

&SCANNE(*ch,str-expr,start*)

The &SCANEQ and &SCANNE functions scan the string value of *str-expr* for the value of *ch*, a single-character string expression. &SCANEQ scans until it finds the same character ("scan until equal"); &SCANNE scans until it finds the first character that is not the same as *ch* ("scan until not equal").

Both functions begin scanning at the character position specified by the integer expression *start*. If *start* is positive, the string is scanned from left to right; if *start* is negative, from right to left. The first character of the string is position 1.

Both functions return an integer value equal to the number of characters skipped before finding the target character (or the first character that is not the target character); that is, they return the value of *index* −&ABS(*start*), where *index* is the position of the character sought. They return special values under these conditions:

- They return a value of 0 if the target character is at the *start* position.
- They return the number of characters in the value of *str-expr* if *start* is 0 or if it is positive and greater than the number of characters in the value of *str-expr*.
- They return a value of 1 if *start* is negative and its absolute value is greater than the number of characters in the value of *str-expr*.

Here are some examples of values returned by &SCANEQ and &SCANNE:

Expression	Result
&SCANNE('.','.....foobar',1)	5
&SCANEQ('.','.....foobar',-11)	-6
&SCANEQ('o','.....foobar',1)	6
&SCANNE('o','.....foobar',7)	2
&SCANNE('.','.....foobar',100)	11
&SCANNE('.','.....foobar',0)	11
&SCANNE('.','.....foobar',-100)	1

&STRTOINT or &S2I: Convert string to integer

{ &STRTOINT | &S2I }(*str-expr*)

The &STRTOINT function accepts a string expression argument and converts it to an integer. If the string cannot be parsed into an integer token, the Assembler will issue a warning and &STRTOINT will return a value of 0. &STRTOINT follows the same token-parsing rules as the Assembler itself; for a discussion of integer tokens, see "Numeric Constants" in Chapter 2 and the discussion of "&LEX" earlier in this chapter. You can also write &STRTOINT as &S2I.

Symbol table functions

A special group of SET functions let you create and manipulate your own symbol tables. They are the following:

&NEWSYMTBL	Creates a new symbol table and returns its id number
&ENTERSYM	Enters or updates a symbol in a symbol table
&FINDSYM	Finds a symbol in a symbol table
&DELSYMTBL	Deletes a symbol table

These functions are described in the next four sections.

&NEWSYMTBL: Create new symbol table

```
&NEWSYMBTBL
```

The &NEWSYMTBL function has no parameters. It creates a new symbol table in memory and returns an integer value that is its ID number. You use this number as the value of *sym-tbl* in subsequent &ENTERSYM, &FINDSYM, and &DELSYMTBL expressions. You may create up to four new symbol tables at any one time, which exist in addition to the Assembler's own local and global tables (identified by the system variables &SYSLOCAL and &SYSGLOBAL). Remember, however, that each symbol table occupies a minimum of 1016 bytes of memory. &NEWSYMBTBL returns a value of zero if it cannot create a new symbol table.

&ENTERSYM: Enter or update symbol in table

&ENTERSYM(*sym-tbl,symbol,value,flags*)

The &ENTERSYM function enters or updates a symbol in the symbol table identified by the integer expression *sym-tbl*. The value of *sym-tbl* may be &SYSLOCAL, &SYSGLOBAL, or any nonzero value returned by &NEWSYMTBL. If it is &SYSLOCAL, you can use &ENTERSYM only in source text inside a code or data module. The value of the string expression *symbol* is the symbol's identifier, with uppercase and lowercase distinctions preserved. If it has already been entered in the table, the existing information is overwritten.

You can use &ENTERSYM to create new symbol table entries or amend existing ones, either in the Assembler's own local or global tables or in a table you have created with &NEWSYMTBL. If you operate on one of the Assembler's tables you must follow its rules, as described below. If you operate on a table you have created, you have more freedom. In particular, you may assign either a string or integer *value* to *symbol*, and may select any 16-bit positive *flags* value.

If the symbol table was created by &NEWSYMTBL, *value* may be either a string expression or a 32-bit integer expression. If it is one of the Assembler's own tables (identified by &SYSLOCAL or &SYSGLOBAL), *value* must be a 32-bit integer. In the latter case, it is used to contain the value assigned to the identifier by an equate directive (see "Symbol Definitions" in Chapter 4).

The integer expression *flags* must have a positive 16-bit value. If *sym-tbl* identifies a table created by &NEWSYMTBL, *flags* may have any value in the range 0..32767. You can create or update EQU or SET definitions in the Assembler's tables with &ENTERSYM, using the appropriate *flags* value.

If *sym-tbl* identifies one of the Assembler's tables, the *flags* value must be 0 if *value* is to be treated as assigned by an EQU directive or 1 if it is to be treated as assigned by a SET directive; no other *flags* values are allowed. &ENTERSYM returns a value of 1 if it has successfully entered or updated the requested symbol table entry; 0 otherwise.

❖ *Note:* Before using &ENTERSYM to enter a new symbol, it is usually wise to check for an existing symbol of the same name by using &FINDSYM.

&FINDSYM: Find symbol in table

&FINDSYM(*sym-tbl,symbol*)

The &FINDSYM function searches for a symbol in the symbol table identified by the integer expression *sym-tbl*. The value of *sym-tbl* may be &SYSLOCAL, &SYSGLOBAL, or any value returned by &NEWSYMTBL. If it is &SYSLOCAL, you can use &FINDSYM only in source text inside a code or data module. The value of the string expression *symbol* is the symbol's identifier, with uppercase and lowercase distinctions preserved.

If &FINDSYM cannot find the symbol, it returns a value of 0. Otherwise it returns a value of 1 and sets the values of the Assembler system variables &SYSVALUE and &SYSFLAGS equal to the *symbol*'s associated values of *value* and *flags*, respectively. If *sym-tbl* identifies a table created by &NEWSYMTBL, these may be any values within the limits described above under "&ENTERSYM," including both strings and integers for the *symbol*'s *value*. If the value of *sym-tbl* is either &SYSLOCAL or &SYSGLOBAL, the following rules apply:

- The value of &SYSFLAGS is 0 for symbols most recently equated by EQU directives and 1 for symbols most recently equated by SET directives,except for register equates.
- The value of &SYSFLAGS is 2 for nonequated symbols and register equates.
- &SYSVALUE contains the symbol's equated value if the value of &SYSFLAGS is 0 or 1; otherwise it contains 0.

&DELSYMTBL: Delete symbol table

`&DELSYMTBL(`*sym-tbl*`)`

The `&DELSYMTBL` function deletes the symbol table identified by the integer expression *sym-tbl*. The value of *sym-tbl* may be any value returned by `&NEWSYMTBL`; it may not be `&SYSLOCAL` or `&SYSGLOBAL`. The Assembler releases all memory allocated for the table, and you should not try to access it again.

`&DELSYMTBL` returns a value of 1 if the table was successfully deleted and 0 otherwise.

SETC and String Expressions

Every string SET variable has a variable-length string value up to 255 characters long. The Macro Processor sets this value to the null string at the first definition of a global string variable and at every definition of a local string variable. You can subsequently change the value using the `SETC` directive. `SETC` statements have this form:

set-var-name `SETC` *str-expr*

The label *set-var-name* must be the identifier of an accessible string SET variable—one that includes the `SETC` directive in its current scope.

The operand *str-expr* may be composed of the following:

- quoted strings
- symbolic parameters of string type
- string SET variables
- string functions

These rules apply to the operand *str-expr*:

- String functions may be combined with other string and integer functions to make more complex string expressions.
- The value of a string expression, including any intermediate values, cannot exceed 255 characters.

The operand *str-expr* may contain any of the following functions, described later in this section:

`&CHR`	Converts an integer value to a one-character string containing its ASCII equivalent
`&CONCAT`	Concatenates string expressions
`&DEFAULT`	Returns a default string value if a test string is null
`&GETENV`	Returns the value of an MPW Shell variable
`&INTTOSTR`	Converts an integer expression to its string value
`&LOWCASE`	Converts a string to all lowercase
`&SETTING`	Returns the current setting value for certain other directives

&SUBSTR	Returns a substring of a string
&TRIM	Removes leading and/or trailing spaces and tabs from a string
&TYPE	Returns the type of a symbol
&UPCASE	Converts a string to all uppercase

Here are some examples of valid string expressions:

'abacab'	String constant
&str	Symbolic parameter or SET variable
&CONCAT(&modname,'.',&var)	Function

Accessing substrings of string variables

When referring to a string variable, including a string SET variable, a string parameter, or a string array element, you may append a **subscript** to access individual characters or substrings in its value. Array elements are described below under "Set Array Variables."

There are two possible forms for a subscript that accesses part of the string value of a SET variable:

[*arith-expr*]	Accesses a single character in the string
[*start*:*length*]	Accesses a substring in the string

Arith-expr, *start*, and *length* may be absolute integer expressions of any complexity. *Arith-expr* accesses a single character, counting the first character of the string as 1. *Start* specifies the position of the first character of a substring, and *length* specifies its length. Here are some examples of accessing parts of the string value of a SET array element; they all assume that the value of the SET variable &foo is the string 'slangword':

Reference	Value
&foo[4]	'n'
&foo[5:3]	'gwo'
&foo[&x:4]	'angw' if &x = 3

The use of a subscript with a string SET variable identifier is subject to these rules:

- When accessing a single character, if the value of *arith-expr* is greater than the length of the string, the Macro Processor will return a string consisting of a single space (ASCII $20) character.
- When accessing a substring, if the value of *start* is zero or is greater than the length of the string, the Macro Processor will return the null string.
- When accessing a substring, if the position of *start* is within the string, but fewer than the number of characters specified by *length* remain, the Macro Processor will return only the remainder of the string.

- You can use the substring form, but not the single character form, when accessing a macro parameter. This is because the Macro Processor interprets a single subscript as identifying an element of a sublist parameter. To access a single character in a macro parameter, use the substring form with a length 1.

&CHR: Convert integer to character

`&CHR(`*arith-expr*`)`

The `&CHR` function returns a string value containing the single character that has the ASCII value specified by *arith-expr*. This value must lie in the range 0..255.

&CONCAT: Concatenate strings

`&CONCAT(`*str-expr*`,...)`

The `&CONCAT` function concatenates the values of any number of string expressions, separated by commas. The value returned by `&CONCAT` is the string resulting from appending all its arguments in the order given. It will be truncated to 255 characters if it exceeds that length.

&DEFAULT: Return string value or default

`&DEFAULT(`*str-expr*$_1$`,`*str-expr*$_2$`)`

The `&DEFAULT` function takes a string expression as its first argument, *str-expr*$_1$. If the value of *str-expr*$_1$ is not null, `&DEFAULT` returns that value. If the value of *str-expr*$_1$ is null, `&DEFAULT` returns the value of the string expression *str-expr*$_2$. Hence you can use *str-expr*$_2$ as a default value for the result.

&GETENV: Return MPW shell variable value

`&GETENV(`*str-expr*`)`

The `&GETENV` function returns the current value of a Macintosh Programmer's Workshop Shell variable specified by the string *str-expr*. If *str-expr* names a variable that is undefined or not exported from the MPW Shell, `&GETENV` returns the null string. MPW Shell variables are explained in the *Macintosh Programmer's Workshop Reference*.

&INTTOSTR or &I2S: Convert integer to string

{ &INTTOSTR | &I2S }(*arith-expr* [,*width* [,*hex*]])

The &INTTOSTR function converts an integer value to its equivalent string form. The first argument, *arith-expr*, specifies the integer to convert. The second argument, *width*, is optional. It specifies the length of the resulting string. The third argument, *hex*, also optional, and used only if *width* is specified, tells the Macro Processor whether to return a hexadecimal or decimal number. Zero produces decimal conversion; any nonzero value produces hexadecimal conversion. &INTTOSTR conforms to these rules:

- If the conversion results in fewer digits than specified by *width*, the left end of the string will be filled with spaces (ASCII $20) unless you specify a negative value for *width*, in which case the Macro Processor will fill the left end of the string with zero characters (ASCII $30).
- If the conversion results in more digits than specified, the Macro Processor will ignore *width* and return a string as long as necessary.
- If you omit the *width* and *hex* arguments, the Macro Processor will assume a *width* of 1 and a *hex* value of 0.

You can also write &INTTOSTR as &I2S.

&LOWCASE or &LC: Convert string to lowercase

{ &LOWCASE | &LC }(*str-expr*)

The &LOWCASE function returns the string expression *str-expr* with any uppercase characters in the range A..Z converted to lowercase. You can also write &LOWCASE as &LC.

&SETTING: Return directive setting

&SETTING(*str-expr*[, *arith-expr*])

The &SETTING function determines the current mode setting for the directive named by the string expression *str-expr*. It returns a string whose value is the name of that mode. By using this function, you can access and store the current setting of a directive before temporarily changing it; this lets you restore the setting later to its original value.

The integer expression *arith-expr* is used only when the value of *str-expr* is 'PRINT'—that is, when &SETTING is used to return values set by the PRINT directive. In this case, *arith-expr* takes a value in the range 0..5 and specifies the age level of PRINT settings that have been stacked by using the directive's PUSH parameter. Zero, the default value, refers to the current setting; five, the maximum value, refers to the setting that held before the last five PRINT PUSH directives. This lets you choose up to six different groups of current or prior PRINT settings; for example, the directive.

```
PRINT      &SETTING('PRINT',3)
```

restores the settings three levels down on the PRINT format stack.

Table 6-2 shows all the directive names that are accepted by &SETTING and the possible values that &SETTING might return for each directive. In the case of PRINT, multiple setting values are concatenated with commas between them.

Table 6-2
&SETTING values

Directive	Possible return values
ALIGN	0, 1
BLANKS	ON, OFF
BRANCH	SHORT, WORD, LONG
CASE	ON, OFF, OBJECT
CODEREFS	FORCEJT, NOFORCEJT, FORCEPC
DATAREFS	ABSOLUTE, RELATIVE
FORWARD	WORD, LONG
MACHINE	MC68000, MC68010, MC68020, MC68030
OPT	ALL, NONE, NOCLR
PRINT	ON, OFF, GEN, NOGEN, PAGE, NOPAGE, WARN, NOWARN, MCALL, NOMCALL, OBJ, NOOBJ, DATA, NODATA, MDIR, NOMDIR, HDR, NOHDR, LITS, NOLITS, STAT, NOSTAT, SYM, NOSYM
STRING	PASCAL, ASIS, C

&SUBSTR: Return substring of string

&SUBSTR(*str-expr,start,length*)

&SUBSTR returns a substring of its first argument, the string expression *str-expr*. Its second and third arguments, *start* and *length*, must be integer expressions. *Start* specifies the position in *str-expr* of the first character of the string returned by &SUBSTR; *length* specifies its length. The first character of *str-expr* is position 1. &SUBSTR follows these rules:

- If the value of *start* is less than one, &SUBSTR returns the null string.
- The value of *length* may not be negative.

- If *start* is greater than the length of *str-expr*, &SUBSTR returns the null string.
- If *length* is greater than the number of characters remaining in *str-expr* after *start*, &SUBSTR will return only the remainder of *str-expr*.

The following are examples of the use of &SUBSTR:

Expression	Result
`&SUBSTR('foobar',4,3)`	`bar`
`&SUBSTR('abcdef',10,2)`	`null`
`&SUBSTR('abcd',3,3)`	`cd`

&TRIM: Trim spaces and tabs from string

`&TRIM(`*str-expr* [, *trim-left*])

The &TRIM function removes leading and trailing spaces and/or tabs (blanks) from the value of the string expression *str-expr*. The argument *trim-left* must be an arithmetic expression. It gives you these possible actions:

- If you omit *trim-left*, &TRIM deletes both leading and trailing blanks.
- If *trim-left* is present and its value is 0, &TRIM deletes only trailing blanks.
- If *trim-left* is present and its value is not 0, &TRIM deletes only leading blanks.

&TYPE: Determine identifier type

`&TYPE(`*str-expr*`)`

The &TYPE function determines the type of a macro or nonmacro identifier, returning a string whose value indicates the type. The identifier is the value of the string expression *str-expr*.

❖ *Note:* If a macro identifier is the same as a nonmacro identifier, &TYPE will return the type of the nonmacro identifier.

The type specification string returned by &TYPE may have various values. The possibilities are listed below. This listing follows these conventions:

- Words enclosed in brackets may or may not be present, depending on the specific type of *str-expr*.
- Words separated by vertical bars (|) represent choices, one of which is always present in the value returned by &TYPE.
- The expression *type* represents a template identifier if *str-expr* refers to an object given a type by a template.
- The expression '['*dim*']' represents the integer value of an array dimension enclosed in brackets.

- The expression '('*type*')' represents the string value of a type identifier enclosed in parentheses.
- UNDEFINED is the word that &TYPE returns if *str-expr* specifies an invalid or undefined identifier.

If *str-expr* specifies a macro variable identifier, &TYPE returns one of the following type descriptions:

```
UNDEFINED

PARM [ STRUCTURED ] { INT | STR }

{ SETA | SETC } [ ARRAY'['dim']' ]

MACRO [{ FUNCTION | SYSVAR }]
```

If *str-expr* specifies a nonmacro identifier, then the identifier is interpreted exactly as if it were used in an absolute expression. It is subject to the standard scope rules given in Chapter 2 under "Scope of Definitions." It may be a fully qualified field reference or a partially qualified reference covered by a WITH directive. With nonmacro identifiers, &TYPE returns one of the following type descriptions:

```
UNDEFINED

{ CODE | DATA } IMPORT

REG { An | Dn | ZAn | ZDn | CCR | SR | USP | MSP | SFC | DFC | CAAR | VBR | CACR | ISP
      CRP | SRP | DRP | TC | PSR | PCSR | AC | CAL | SCC | VAL | BADn | BACn }

FPREG {FPn | FPCR | FPSR | FPIAR }

RLIST

FRLIST

FCRLIST

{ CODE | DATA } MODULE { EXPORT | ENTRY | IMPORT } [ MAIN ] [ '('type')' ]

TEMPLATE [ DATA IMPORT ] [ '('type')' ]

TEMPLATE FIELD [ '('type')' ]

DATA FIELD [ { EXPORT | ENTRY | IMPORT } ] [ '('type')' ]

{ CODE | DATA } LABEL [ { EXPORT | ENTRY | IMPORT } ] [ MAIN ] [ '('type')' ]

SET

EQU

OPWORD
```

&UPCASE or &UC: Convert string to uppercase

{ &UPCASE | &UC }(*str-expr*)

The &UPCASE function returns *str-expr* with any lowercase characters in the range a..z converted to uppercase. You can also write &UPCASE as &UC.

SET array variables

You can define a SET variable as an array, thereby referring to many values with a single identifier. The individual values in a SET array variable are called **elements.** SET array elements may contain either integer or string values and may have either local or global scope. To access an element in a SET array variable, follow the SET variable identifier immediately with a subscript enclosed in brackets.

SET array variables are subject to these rules:

- All elements in a variable must have the same type and scope.
- The subscript may be any integer expression, as long as its value is not 0 or negative.
- You may not use a subscript with a nonarray SET variable.
- A SET array element cannot be accessed without a subscript.

The following are examples of valid SET array element references:

```
&foo[12]
&foo[&bar]
&foo[12+&bar]
&foo[&NBR(&bar)]
```

The following are examples of erroneous set array element references.

`&foo`	No subscript (a good nonarray variable reference)
`[12]`	No SET variable identifier
`&foo [12]`	Space between identifier and subscript

You may not continue the subscript part of a SET array variable reference to the next line.

Defining SET array variables

You define SET array variables the same way as other SET variables, except that you enclose a decimal number in brackets immediately following the variable's identifier in the variable list of your LCLA, LCLC, GBLA, or GBLC directive. This decimal number is called the **dimension;** it specifies the number of elements in the array. A SET array variable may not have more than one dimension number. The maximum dimension number allowed is 4096. Arrays with dimensions larger than 250 are represented as sparse arrays, and do not necessarily have memory allocated for unassigned elements. You may mix SET array variable definitions with nonarray definitions in the same directive.

If you define a SET array variable as global, you must specify the same dimension number every time you define it again as global. Alternately, you can write an asterisk (*) in place of the dimension number after the first definition; this indicates that the first dimension number is to be used. The first time a SET array variable is defined, the Macro Processor assigns to each of its elements either 0 or the null string, depending on whether the variable's type is integer or string.

Following are examples of valid definitions of SET array variables:

```
            LCLA        &foo[10]
            LCLC        &bar[100],&car[54]
            GBLA        &totals[4]
            GBLC        &currline,&lines[100],&charset[256]
```

Using SET array variables

Once you have defined a SET array variable, you may assign values to its elements with SETA or SETC directives. You can then access these values by referring to the array's elements in subsequent statements or directives. The subscripts in references to SET array elements may be arithmetic expressions of any complexity, as well as explicit decimal integers.

The following example illustrates the definition and use of a SET array variable:

```
            MACRO
            ArgScan                             ; No parameters, uses &&SYSLIST
            LCLC        &a[10]
            LCLA        &i,&n
&n          SETA        &NBR(&SYSLIST)          ; Number of actual macro parameters
&i          SETA        1
            WHILE       &i<=&n DO
&a[&i]      SETC        &SYSLIST[&i]
&i          SETA        &i+1
            ENDW
.* the macro parameters have now been stored in the &a array,
```

```
.* where they can be manipulated or modified. The macro can
.* always access their original values via &SYSLIST
          - - -                                   ; Body of macro
          ENDM
```

Accessing substrings in SET array string elements

When referring to a SET array element that is a string, you may append a second subscript to access individual characters or substrings in the string.

The section "SETC and String Expressions," earlier in this chapter explained how a subscript on a nonarray SET variable of type string lets you access parts of the string value. In the same way, a second subscript on an SET array variable with string elements extracts part of the array string specified by the first subscript.

There are two possible forms for a subscript that accesses part of a string:

'[' *array-sub,arith-expr* ']' Accesses a single character in the string

'[' *array-sub,start:length* ']' Accesses a substring in the string

Array-sub is the array subscript. *Arith-expr*, *start*, and *length* may be integer expressions of any complexity. *Arith-expr* accesses a single character, counting the first character of the string as 1. *Start* specifies the position of the first character of a substring, and *length* specifies its length.

Here are some examples of accessing parts of a SET array element with string value; they all assume that the value of the sixth element of the SET array variable `&foo` is the string `'slangword'`:

Reference	Value
`&foo[6,4]`	`'n'`
`&foo[6,5:3]`	`'gwo'`
`&foo[x+3,x:4]`	`'angw'` if `x = 3`

The use of a second subscript with a SET string array variable identifier is subject to these rules:

- When accessing a single character, if the value of *arith-expr* is greater than the length of the string, the Macro Processor will return a string consisting of a single space character (ASCII $20).
- When accessing a substring, if *start* is zero or is greater than the length of the string, the Macro Processor will return the null string.
- When accessing a substring, if the position of *start* is within the string but fewer than *length* characters remain, the Macro Processor will return only the remainder of the string.

- You can use the substring form, but not the single character form, when accessing a macro parameter sublist. To access a single character in a macro parameter sublist, specify a substring of length 1.

Assembler system variables

Assembler system variables are variables which are defined and assigned values by the Assembler. These values may change during the assembly process. If you define them in your source text, you will override their original purpose. They are also subject to the restrictions given for SET variables earlier in this chapter. Table 6-3 lists the Assembler system variables.

Table 6-3
Assembler system variables

Name	Value	Type
`&SYSNDX`	Index number of the current macro call	Integer or string
`&SYSLIST`	Current macro parameter list	String
`&SYSSEG`	Identifier of the current code segment	String
`&SYSMOD`	Identifier of the current code module	String
`&SYSDATE`	Current date	String
`&SYSTIME`	Current time	String
`&SYSTOKEN`	Token code from &LEX call	Integer
`&SYSTOKSTR`	Token string from &LEX call	String
`&SYSVALUE`	Symbol value returned by &FINDSYM	Integer or string
`&SYSFLAGS`	Symbol flags returned by &FINDSYM	Integer
`&SYSLOCAL`	ID of the Assembler's local symbol table	Integer
`&SYSGLOBAL`	ID of the Assembler's global symbol table	Integer

These variables are further described in the sections below.

&SYSNDX or &SYSINDEX: Macro call index

The Macro Processor assigns `&SYSNDX` the four-digit number 0001 when it encounters the first macro call directive in your source text. It increases this number by one at each subsequent macro call directive, including inner macro calls. The value of &SYSNDX remains the same throughout macro expansion (being restored after inner macro calls). You can also write `&SYSNDX` as `&SYSINDEX`.

If &SYSNDX is used in a model statement or in a string expression, the Macro Processor treats it as a string symbol. Its value will include any leading zeros needed to create a four-digit number. If &SYSNDX is used in an integer expression (for example, in the operand of a SETA directive), the Macro Processor treats it as an ordinary integer.

A typical use of &SYSNDX is to concatenate it with other characters, thereby creating unique labels for statements generated during different expansions of the same macro model statement.

&SYSLIST or &SYSLST: Macro operand list

The &SYSLIST variable gives you an alternate way to access macro parameters. However, it accesses only the positional parameters; it does not access keyword parameters, described in Chapter 5. You can also write &SYSLIST as &SYSLST.

You can use &SYSLIST only in macro definitions.

&SYSLIST[*n*] refers to the nth positional operand of the current macro. If the *n*th operand is a sublist, &SYSLIST[*n*,*m*] refers to the *m*th sublist element of the *n*th operand. You may use any absolute integer expression for the subscripts *n* and *m*, as long as their values are greater than zero and not greater than the numbers of parameters and/or sublist elements. If the *m*th element of the *n*th parameter is itself a sublist, then &SYSLIST[*n*,*m*,*k*] refers to its *k*th element, and so on.

You can use the &NBR function with &SYSLIST to determine the number of positional operands that were specified when the current macro was called. When applied to &SYSLIST[*n*], &NBR returns these values:

- the number of elements in the *n*th operand, if it is a sublist
- 0 if the *n*th operand is not a sublist
- 0 if the *n*th operand was omitted in the current macro call

Here are some examples of values returned by &SYSLIST expressions in a macro definition whose parameter list is A,(B,C),D:

Expression	Value
&SYSLIST[1]	A
&SYSLIST[2]	(B,C)
&SYSLIST[3]	D
&SYSLIST[2,1]	B
&SYSLIST[2,2]	C

&SYSSEG: Current segment identifier

The value of the `&SYSSEG` variable is the name of the current code segment.

&SYSMOD: Current module identifier

The value of the `&SYSMOD` variable is the identifier of the current module. This value is the null string if `&SYSMOD` is used outside a module or inside an unnamed module.

&SYSDATE: Current date

The value of the `&SYSDATE` variable is the current date, expressed in the the following format:

dd-mmm-yy

dd = String containing two-numeral day of the month

mmm = String containing first three letters of the current month

yy = String containing two-numeral year

All these values are copied from the Macintosh internal clock.

&SYSTIME: Cu rent time

The value of the `&SYSTIME` variable is the current time, expressed in the the following format:

hh:mm:ss

hh = String containing two-numeral hour of the day (24-hour basis)

mm = String containing two-numeral minute

ss = String containing two-numeral second

All these values are copied from the Macintosh internal clock.

&SYSTOKEN and &SYSTOKSTR: Values set by &LEX

The values of &SYSTOKEN and &SYSTOKSTR are set every time &LEX is called. &SYSTOKEN contains an integer code indicating the token type, and &SYSTOKSTR contains the token's string value. Table 6-1 shows all possible values of these variables.

&SYSVALUE and &SYSFLAGS: Values set by &FINDSYM

The values of &SYSVALUE and &SYSFLAGS are set every time &FINDSYM is called. &SYSVALUE contains the value associated with the symbol found, and &SYSFLAGS contains a positive 16-bit integer that indicates certain characteristics of the symbol. These variables have different types and values, depending on how &FINDSYM was called. For further information, see "&FINDSYM" earlier in this chapter.

&SYSLOCAL and &SYSGLOBAL: System symbol table ID's

These variables have fixed values assigned by the Assembler. They are used only as parameters for the &ENTERSYM and &FINDSYM functions. &SYSLOCAL may be used only in source text inside a code or data module. For further information, see "&ENTERSYM" and "&FINDSYM" earlier in this chapter.

Chapter 7

Macro and Conditional-Assembly Directives

The power of the macro language depends primarily on its ability to loop and branch, thereby letting you program the ways that it expands your original source text. This section discusses the following directives:

GOTO	Unconditional jump to another part of the source text
IF...GOTO	Conditional jump to another part of the source text
IF...ENDIF	Conditional assembly of enclosed source text
ELSEIF, ELSE	Further conditions on assembly of enclosed source text
WHILE...ENDWHILE	Conditional looping of enclosed source text
ACTR	Set maximum number of branches in macro expansion
EXITM	Unconditional termination of macro
WRITE, WRITELN	Write information to the diagnostic output
AERROR	Generate an Assembler error
ANOP	Assembler NOP

Many of these directives use Boolean expressions for control. Such expressions are discussed in the next section.

Boolean control expressions

Conditional-assembly directives are controlled by Boolean expressions, in which integer values are treated as true or false. Hence every absolute integer expression may also be used as a Boolean expression. The Macro Processor interprets an integer value of 0 as a Boolean value of false; it interprets any nonzero integer value as a Boolean value of true. For correct results with expressions containing AND or OR, however, you should always use 1 to denote the Boolean value true.

In addition, however, Boolean expressions may be constructed from **logical** and **comparison operators.** To extend the utility of these operators in macro constructs, the comparison operators may be used for string comparisons as well as for integer comparisons.

The logical operators are OR, XOR, AND and NOT. The comparison operators are =, <> or ≠, >, <, >= or ≥, and <= or ≤. The syntax rules for writing these operators are discussed more fully under "Expressions" in Chapter 2.

Each operand of a comparison expression may have either an integer or a string value. The ways the Macro Processor evaluates the whole expression depends on the types of the two operands, as discussed below.

Comparing two integer expressions

The Macro Processor performs integer comparisons in the normal arithmetic way; the results are always either zero (false) or one (true).

Comparing two string expressions

String comparisons compare the values of two string expressions for equality, inequality, and relative alphabetical ordering. The results are integer values—zero for false, one for true.

The Macro Processor distinguishes between uppercase and lowercase when comparing strings. You can mask the distinction by using the &UPCASE or &LOWCASE functions described under "SETC and StringExpressions" in Chapter 6.

Two strings are **equal** if they are indistinguishable; otherwise they are **unequal.**

The Macro Processor ranks strings by **relative ASCII ordering** for comparisons using the >, <, ≥, and ≤ operators. It performs this ranking by the following steps:

1. The two strings are compared a character at a time, starting with the first character
2. Two corresponding characters are compared. If the ASCII value of one character is greater than the other, then the corresponding string is greater than the other.
3. If the two corresponding characters are equal, the point of comparison advances to the next character in each string, and the process returns to step 2.
4. If the end of one string is reached before the end of the other, its value is less than the other string.
5. If the ends of both strings have been reached, the two strings are equal.

Comparing integer and string expressions

To compare an integer with a string, the Macro Processor first converts the second operand to the type of the first operand. If the first operand is type integer, the second operand will be converted to an integer by interpreting the string contents as an integer constant. It does this by following the same parsing rules as &ISINT, described "SETA and StringExpressions" in Chapter 6. If the string does not contain an integer, the Assembler reports an error. If the first operand is a string, the integer operand will be converted to a string containing the digits that represent the operand's value. Note that comparisons of the string representations of integers do not always yield the same results as arithmetic comparison of the same integers.

GOTO, IF...GOTO, and macro labels: branching

[*macro-label*] GOTO [+ | –]{ *macro-label* | *str-expr* }

[*macro-label*] IF *bool-expr* GOTO [+ | –]{ *macro-label* | *str-expr* }

You can use the GOTO and conditional GOTO directives to alter the sequence in which the Assembler processes your source text. The destination of a GOTO directive is always a **macro label**. You can specify a macro label by using either a macro label identifier or an expression whose value is a string containing a macro label identifier.

A macro label consists of a period followed by a valid identifier, and is defined when it appears in the label field of a statement or directive. If the period is not the first character of the source text line, you must terminate the macro label with a colon. You may write macro labels in any machine instruction statement, or in any directive statement that accepts a macro label. Macro labels conform to these rules:

- Macro labels defined within a macro are local to that macro.
- Macro labels with the same identifier are not allowed in the same scope.
- A macro label may appear on a line by itself, without an operation. This is necessary in cases where the destination line already has a label. Alternatively, you can write a label on an ANOP directive.
- The Macro Processor does not distinguish between uppercase and lowercase when interpreting macro labels.

If the jump is within a macro, you may optionally write either a plus or a minus character before the macro label to tell the Macro Processor whether the jump is forward or backward. Doing this decreases assembly time.

The GOTO directive causes assembly processing to jump unconditionally to the source text line where the specified label is defined. Any lines passed over are not processed.

The IF...GOTO directive determines the value of the expression *bool-expr*. If it is true (nonzero), the assembly process jumps to the source text line where the specified label is defined. If it is false (zero), processing continues with the succeeding source text line as usual.

When using GOTO and IF...GOTO, bear these rules in mind:

- GOTO and IF...GOTO directives outside macros may jump only to macro labels that are defined in subsequent statements; backward jumps are not allowed outside macros.
- GOTO and IF...GOTO directives in a macro definition may not cause a branch to any label outside the body of that macro except to the macro label (if any) in the macro's ENDM statement.

The following example illustrates the use of GOTO directives and macro labels. It uses a conditional GOTO directive to avoid the unnecessary code sometimes generated by the sample macro example given in Chapter 5 under "Symbolic Parameters."

```
            MACRO
            Incr        &src,&dest
            IF          UPCASE(&src)='D0' GOTO .lab1
            MOVE.W      &src,D0
.lab1       ADDQ.W      #1,D0
            IF          UPCASE(&dest)='D0' GOTO .lab2
            MOVE.W      D0,&dest
.lab2       ENDM
```

IF, ELSEIF, ELSE, and ENDIF: Conditional Assembly

[*macro-label*] IF *bool-expr* THEN

statements

ELSEIF *bool-expr* THEN

statements

ELSE

statements

ENDI[F]

The IF, ELSEIF, ELSE, and ENDIF directives provide facilities for conditional assembly. You use them to enclose sections of source text that the Assembler will process only if certain Boolean values are true. You may abbreviate ENDIF as ENDI.

Each IF directive must be followed by an ENDIF directive. Together they form an **IF...ENDIF construct**. ELSEIF and ELSE directives are optional, but when they appear they must be located within an IF...ENDIF construct, as indicated in the above syntax diagram. Here are the rules:

- You can write any number of ELSEIF directives, but only one ELSE directive.
- The ELSE directive, if any, must follow all the ELSEIF directives.
- The region of source text controlled by IF extends to its corresponding ENDIF, or to the first ELSEIF or ELSE, whichever comes first.
- The region of source text controlled by each ELSEIF extends to the next ELSEIF , ELSE, or ENDIF, whichever comes first.
- The region of source text controlled by ELSE extends to ENDIF.

When the Macro Processor processes IF and ELSEIF directives, it evaluates their Boolean expressions. For a given `IF...ENDIF` construct, the first Boolean value of true in an IF or ELSEIF directive will cause assembly of the source text controlled by that directive. All further ELSEIF directives in the construct, and the source text they control, will be skipped. If the Macro Processor does not find any true IF or ELSEIF directives in the construct, it assembles the source text controlled by ELSE. Remember that an entire `IF...ENDIF` construct may be skipped if it is enclosed in an `IF...ENDIF` or `WHILE...ENDWHILE` construct whose Boolean control value is false.

You can nest `IF...ENDIF` constructs; the Macro Processor will associate the last IF with the first ENDIF, the next-to-last IF with the second ENDIF, and so on.

The following example illustrates the use of an `IF...ENDIF` construct. It improves the example given in Chapter 5 under "Macro Comment" by generating a debug procedure header only if the global SET variable &Debugging is `true` (nonzero).

```
MACRO
DbgHead
GBLA        &Debugging          ; Declare (import) global variable
IF          &Debugging THEN     ; Header only if debugging
LINK        A6,#0               ; Debug header
ENDIF
ENDM
```

WHILE and ENDWHILE: Looping

[*macro-label*]	`WHILE` *bool-expr* `DO`
	statements
[*macro-label*]	`ENDW[HILE]`

The WHILE and ENDWHILE directives provide conditional looping. You can use them only in macro definitions. You use them in pairs to enclose sections of source text that the Assembler will process repeatedly, as long as the value of *bool-expr* is true. The result is called a **`WHILE...ENDWHILE` construct.** You may abbreviate ENDWHILE as ENDW.

When the Macro Processor encounters a WHILE directive, it evaluates its Boolean expression. If the value of the expression is true (nonzero), the Macro Processor processes source text until the corresponding ENDWHILE. If the Boolean value is false (zero), the Macro Processor skips to the source text line immediately following ENDWHILE.

When the Macro Processor encounters ENDWHILE (which happens only if the corresponding WHILE directive was `true`), the next line it processes will be the corresponding WHILE directive. In other words, it loops back from ENDWHILE to WHILE. It evaluates the WHILE directive's Boolean expression again and repeats the procedure described above.

WHILE...ENDWHILE constructs may be nested; the Macro Processor will associate the last WHILE with the first ENDWHILE, the next-to-last WHILE with the second ENDWHILE, and so on.

ACTR: Limit looping

[*macro-label*] ACTR *arith-expr*

The ACTR directive sets the maximum number of GOTO, IF...GOTO, and WHILE branches that the Macro Processor can make. This number is the value of *arith-expr*. If you write an ACTR directive in the body of a macro definition, the limit applies to each expansion of that macro. If you write it anywhere else, it applies to the whole assembly process except macro expansions. The limit remains in force until the end of the assembly process or until it is superseded by another ACTR directive. In the absence of any ACTR directive, the preset limit value is 512.

The ACTR directive is useful to prevent macros that have not been debugged from becoming processed in infinite loops.

Here are some rules about ACTR:

- The value of *arith-expr* must be positive.
- When counting branches during macro expansion, ACTR ignores branches made during inner macro calls.
- Any time the actual number of branches exceeds the limit set by ACTR, the Assembler reports an error.
- If the limit set by ACTR is exceeded during a macro expansion, that macro expansion will terminate.
- If the limit set by ACTR is exceeded outside any macro expansion (which could be caused only by forward GOTO branches), the entire assembly process will terminate.

EXITM or MEXIT: Exit macro

[*macro-label*] EXITM

The EXITM directive terminates expansion of the macro currently being called. You can use EXITM only within the body of a macro definition. You can also write EXITM as MEXIT.

After an EXITM directive is processed, the next source text line to be assembled is the one just after the macro call that invoked the macro just terminated. If that macro call is an inner macro call (nested inside another macro) then the Macro Processor continues processing the higher-level macro.

❖ *Note:* Don't confuse EXITM with ENDM, which is the last directive in every macro definition.

WRITE and WRITELN: Write to the diagnostic output file

[*macro-label*] WRITE *item*,...

[*macro-label*] WRITELN [*item*],...

The WRITE and WRITELN directives write information to the diagnostic output file during the assembly process. You may use them both within macros and outside macros.

The *item* operands may be symbolic parameters, SET variables, string expressions, or integer expressions. The Assembler will send their values to the diagnostic output file with no intervening spaces. The difference between WRITE and WRITELN is that WRITELN sends a final return character (ASCII $0D) after the last item. WRITELN with no operand sends a single return character to the diagnostic output.

WRITE and WRITELN are typically used for debugging macros or for displaying information about the current state of the assembly process.

AERROR: Error Generation

[*macro-label*] AERROR *str-expr*

The AERROR directive generates a synthetic Assembler error, and sends the value of the string expression *str-expr* to the diagnostic output as the error message. You can use AERROR both within macro definitions and outside macros.

AERROR is typically used in conjunction with a conditional directive to report when conditions required for a successful assembly or macro expansion have not been met.

Here is an example:

```
AERROR          'Bad parameter value to macro foo'
```

ANOP: Assembler NOP

[macro-label] ANOP

The ANOP directive lets you specify a macro label as the destination of a GOTO directive. The macro-label destination of a GOTO directive must be placed in the label field of the statement to which it branches. If there is already a symbol in the label field (for example, a SETA assignment) you cannot put the macro label there. However, you can achieve the same effect by labelling an ANOP statement just before the desired destination statement.

❖ *Note:* Using ANOP is not the quite the same as defining a macro label on a line with a blank operation field. ANOP does not generate a line from the macro, while a macro label on a line by itself will generate a blank line. This difference shows up on an assembly listing.

Appendix A

Generic Instruction Formats

You can write certain assembly-language instructions in generic form, letting the Assembler transform them automatically to the final instructions that will appear in your source text. There are three general reasons for writing generic instructions:

- Optimization: The Assembler transforms your instructions if they can be encoded more efficiently. The result occupies less memory and often runs faster as well.
- Convenience: The Assembler transforms your instructions on the basis of their context, to make coding easier and your source text more readable.
- Compatibility: The Assembler transforms your instructions to make them compatible with other assemblers. Other assemblers are discussed in Appendix E.

The OPT directive described in Chapter 4 under "Assembly Options" gives you some control of whether or not the Assembler performs address optimization and generic instruction substitution.

Whenever the Assembler performs a generic instruction substitution, it identifies that statement in the Assembly listing with a letter G in the flags column. See Appendix C for a full description of the Assembly listing format.

For each instruction that you can write generically, Table A-1 lists the following:

The table is grouped by reason for the transformation

- the generic form that you write
- the form generated by the Assembler
- any conditions that must be met for the Assembler to perform the transformation

Table A-1
Generic Instruction Conversions

Generic instruction		Assembled form		Conditions
Optimization				
ADDA	#data,<ea1>	ADDQ	#data,<ea1>	#data=1..8
CLR.L	Dn	MOVEQ	#0,Dn	
MOVE	#0,<ea3>	CLR	<ea3>	<ea3> ≠ Dn
MOVE.L	#data,An	MOVEA.W	#data,An	#data=-32768..32767
MOVE.L	#data,Dn	MOVEQ	#data,Dn	#data=-128..127
MOVE	#0,An	SUBA	An,An	
MOVEA	#0,An	SUBA	An,An	
MOVEA.L	#data,An	MOVEA.W	#data,An	#data=-32768..32767
MOVEQ	#0,An	SUBA	An,An	
SUB	#data,<ea1>	SUBQ	#data,<ea1>	#data = 1..8
SUBA	#data,<ea1>	SUBQ	#data,<ea1>	#data = 1..8
Convenience				
ADD	#data,<ea3>	ADDI	#data,<ea3>	
ADD	<ea0>,An	ADDA	<ea0>,An	
AND	#data,<ea3>	ANDI	#data,<ea3>	
BNZ	label	BNE	label	
Bcc.S	label	NOP	label	label = next instruction
BT	label	BRA	label	
BZ	label	BEQ	label	
CMP	#data,<ea3>	CMPI	#data,<ea3>	
CMP	<ea3>,An	CMPA	<ea3>,An	
CMP	(An)+,(An)+	CMPM	(An)+,(An)+	
DBNZ	Dn,label	DBNE	Dn,label	
DBZ	Dn,label	DBEQ	Dn,label	
DBRA	Dn,label	DBF	Dn,label	
EOR	#data,<ea3>	EORI	#data,<ea3>	
MOVE	<ea0>,An	MOVEA	<ea0>,An	
MOVEM	<ea8>,An	MOVEA	<ea8>,An	
OR	#data,<ea3>	ORI	#data,<ea3>	
SNZ	<ea3>	SNE	<ea3>	
SUB	#data,<ea3>	SUBI	#data,<ea3>	
SUB	<ea0>,An	SUBA	<ea0>,An	
TNZ		TNE		
TZ		TEQ		
TPNZ	#data	TPNE	#data	
TPZ	#data	TPEQ	#data	

Table A-1 (continued)
Generic Instruction Conversions

Generic Instruction		Assembled form		Conditions
Compatibility				
BHS	label	BCC	label	
BLO	label	BCS	label	
DBHS	Dn,label	DBCC	Dn,label	
DBLO	Dn,label	DBCS	Dn,label	
EXTB	Dn	EXT.W	Dn	
EXTW	Dn	EXT.L	Dn	
SHS	<ea3>	SCC	<ea3>	
SLO	<ea3>	SCS	<ea3>	
THS		TCC		
TLO		TCS		
TPHS	#data	TPCC	#data	
TPLO	#data	TPCS	#data	

Appendix B

Syntax Diagrams

This appendix groups together all the syntax diagrams used in this book. The listing is divided into the following sections:

- assembly-language addresses (described in Chapter 3)
- special address formats (described in Chapter 3)
- general assembly directives (described in Chapter 4)
- macro and SET variable directives (described in Chapter 5)
- SET variable functions (described in Chapter 5)

For an explanation of how to interpret these diagrams, see "Notation Conventions" in the Preface.

Assembly-language addresses

For an explanation of the symbols used in address syntax diagrams, see Table 3-1.

Addressing modes

ID	Mode	Effective address syntax
0	Data register direct	D*n*
1	Address register direct	A*n*
2	Address register indirect	(A*n*)
3	Postincrement register indirect	(A*n*)+

4	Predecrement register indirect	–(A*n*)		
5	Indirect with 16-bit displacement	*d*(A*n*)		
6	Indirect with indexing plus 8-bit displacement	*d*(A*n*,Xn)		
6	Indirect with indexing plus base displacement	(*bd*,A*n*,*Xn*s*)	*bd*(A*n*,*Xn*s*)	
6	Indirect with preindexing	([*bd*,A*n*,*Xn*s*],*od*)		
6	Indirect with postindexing	([*bd*,A*n*],*Xn*s*,*od*)		
70	Absolute word (16 bits)	*ae*	(*ae*).W	
71	Absolute long (32 bits)	*ae*	(*ae*).L	
72	PC-relative with 16-bit displacement	*re*	*d*(PC)	
73	PC-relative, indexing, 8-bit displacement	*d*(D*n*)	*d*(PC,*Xn*)	
73	PC-relative, indexing, base displacement	(*bd*,PC,*Xn*s*)	*bd*(PC,*Xn*s*)	*bd*(*Xn*s*)
73	PC-relative with preindexing	([*bd*,PC,*Xn*s*],*od*)		
73	PC-relative with postindexing	([*bd*,PC],*Xn*s*,*od*)		
74	Immediate	#*ae*		
72	Literal (PC-relative with 16-bit displacement)	#*ae*	#(*ae*).W	#(*ae*).L

Address optimizations

Original form	Condition for optimization	Optimized form
(*bd*,A*n*,*Xn*s*)	size of *bd* ≤ 8 bits	*bd*(A*n*,*Xn*s*)
(A*n*,*Xn*s*)	omitted *bd* (*bd* = 0)	0(A*n*,*Xn*s*)
(*bd*,PC,*Xn*s*)	size of *bd* ≤ 8 bits	*bd*(PC,*Xn*s*)
(*bd*,A*n*)	size of *bd* ≤ 16 bits	*bd*(A*n*)
(*bd*,PC)	size of *bd* ≤ 16 bits	*bd*(PC)
d(A*n*)	*d* = 0	(A*n*)

Special address formats

MC68000 instructions

MOVEM: Multiple moves

MOVEM.*size* *rlist,ea*

MOVEM.*size* *ea,rlist*

size ::= W | L

MC68020 instructions

MULS and MULU: Signed and unsigned multiplication

MULS.L	*ea*,D*l*	32 x 32 --> 32
MULS.L	*ea*,D*h*:D*l*	32 x 32 --> 64
MULU.L	*ea*,D*l*	32 x 32 --> 32
MULU.L	*ea*,D*h*:D*l*	32 x 32 --> 64

DIVS and DIVU: Signed and unsigned division

DIVS.L	*ea*,D*q*	32/32 --> 32q
DIVS.L	*ea*,D*r*:D*q*	64/32 --> 32r:32q
DIVU.L	*ea*,D*q*	32/32 --> 32q
DIVU.L	*ea*,D*r*:D*q*	64/32 --> 32r:32q

TDIVS and TDIVU: Truncated signed and unsigned division

TDIVS.L	*ea*,D*q*	32/32 --> 32q
TDIVS.L	*ea*,D*r*:D*q*	32/32 --> 32r:32q
TDIVU.L	*ea*,D*q*	32/32 --> 32q
TDIVU.L	*ea*,D*r*:D*q*	32/32 --> 32r:32q

PACK and UNPK: Packing and unpacking

PACK	–(A*x*),–(A*y*),#*adjustment*
PACK	D*x*,D*y*,#*adjustment*
UNPK	–(A*x*),–(A*y*),#*adjustment*
UNPK	D*x*,D*y*,#*adjustment*

CAS and CAS2: Comparing and swapping

CAS.*size*	D*c*,D*u*,*ea*
CAS2.*size*	D*c1*:D*c2*,D*u1*:D*u2*,(*Rn1*):(*Rn2*)

size ::= B | W | L

Bit field instructions

BFCHG	*ea*{*offset*:*width*}
BFCLR	*ea*{*offset*:*width*}
BFEXTS	*ea*{*offset*:*width*},D*n*
BFEXTU	*ea*{*offset*:*width*},D*n*
BFFFO	*ea*{*offset*:*width*},D*n*
BFINS	D*n*,*ea*{*offset*:*width*}
BFSET	*ea*{*offset*:*width*}
BFTST	*ea*{*offset*:*width*}

Tcc and TPcc: Trap on condition

T*cc*

TP*cc.size* #*ae*

size ::= W | L

MC68881 instructions

FMOVEM with explicit register lists

FMOVEM.*size* *fp-rlist,ea*

FMOVEM.*size* *ea,fp-rlist*

size ::= L | X

FMOVE with packed BCD data

FMOVE.P FP*n,ea*

FMOVE.P FP*n,ea*{#*k*}

FMOVE.P FP*n,ea*{D*n*}

FSINCOS: Simultaneous sine and cosine

FSINCOS.*size* *ea*,FP*c*:FP*s*

FSINCOS.X FP*m*,FP*c*:FP*s*

size ::= B | W | L | S | D | X | P

FTcc and FTPcc: Floating-point trap on condition

FT*cc*

FTP*cc.size* #*ae*

size ::= W | L

FTEST: Test operand and set floating-point condition codes

FTEST.*size* *ea*

FTEST.X FP*n*

size ::= B | W | L | S | D | X | P

MC68851 instructions

Opcode	Operand format	Sizes
PB*cc.size*	*label*>	W | L
PDB*cc.size*	Dn,*label*>	W
PFLUSH	*fc*,#*ae*[,*ea*]	
PFLUSHA		
PFLUSHS	*fc*,#*ae*[,*ea*]	
PFLUSHR	*ea*	D
PLOADR	*fc*,*ea*	
PLOADW	*fc*,*ea*	
PMOVE	*PMMU-reg*,*ea*	B|W|L|D
PMOVE	*ea*,*PMMU-reg*	B|W|L|D
PRESTORE	*ea*	
PSAVE	*ea*	
PS*cc*	*ea*	B
PTESTR	*fc*,*ea*,#*ae*[,A*n*]	
PTESTW	*fc*,*ea*,#*ae*[,A*n*]	
PT*cc*		
PTP*cc*	#*ae*	W | L
PVALID	VAL,*ea*	L
PVALID	A*n*,*ea*	L
fc ::=	#*ae*	(specified as 4 bits in the command word)
D*n*	(contained in the lower 4 bits of Dn)	
SFC	(contained in the processor's source function register)	
DFC	(contained in the processor's destination function code register)	

Literals

```
PEA     #(ae).W
PEA     #(ae).L
PEA     #(ae).S
PEA     #(ae).D
PEA     #(ae).X
PEA     #(ae).P
LEA     #(ae).W,An
LEA     #(ae).L,An
LEA     #(ae).S,An
LEA     #(ae).D,An
LEA     #(ae).X,An
LEA     #(ae).P,An
```

General assembly directives

```
[macro-label]   ALIGN       [expr]

[macro-label]   BLANKS      { ON  | Y[ES] }
                            { OFF | N[O]  }

[macro-label]   BRANCH      { S[HORT] | B[YTE] }
                            { W[ORD]           }
                            { L[ONG]           }

[macro-label]   CASE        { ON  | Y[ES]  }
                            { OFF | N[O]   }
                            { OBJ[ECT]     }

[macro-label]   CODE

[macro-label]   CODEREFS    { F[ORCE[JT]]      }
                            { NOF[ORCE[JT]]    }
                            { F[ORCE]PC        }
```

[macro-label]	COMMENT	string
[macro-label]	DATA	[{ INCR[EMENT] | DECR[EMENT] | MAIN }]
[macro-label]	DATAREFS	{ R[EL[ATIVE]] | A[BS[OLUTE]] }
[{label | macro-label}]	DC[.size]	{ expr | string } , . . .
[{label | macro-label}]	DCB[.size]	length,{ expr | string }
[{ *label* | *macro-label* }]	DS[.*size*]	{ *length* | *template-name* }
[macro-label]	DUMP	filename
[macro-label]	EJECT	[lines]
[macro-label]	END	
[macro-label]	ENDF[UNC]	
[macro-label]	ENDMAIN	
[macro-label]	ENDP[ROC]	
[macro-label]	ENDR	
[macro-label]	ENDWITH	
[macro-label]	ENTRY	{ (name$_1$,name$_2$, . . .) : { CODE | DATA } | name$_1$ [: { CODE | DATA | MAIN }] , . . . } , . . .

name	EQU	{ *arith-expr* | *reg* | *import-name* }
[macro-label]	ERRLOG	filename
[macro-label]	EXPORT	{ (name$_1$,name$_2$, . . .) : { CODE | DATA } | name$_1$ [: { CODE | DATA | MAIN }] , . . . } , . . .
[macro-label]	FORWARD	{ W[ORD] | L[ONG] }
name	FREG	fp-rlist
[name]	FUNC	[{ ENTRY | EXPORT }]
[macro-label]	IMPORT	{ (name$_1$,name$_2$, . . .) : { CODE | DATA | type } | name$_1$ [: { CODE | DATA | type }] , . . . } , . . .
[macro-label]	INCLUDE	filename
[macro-label]	LOAD	filename
[macro-label]	MACHINE	{ MC68000 | MC68010 | MC68020 | MC68030 }
[name]	MAIN	[{ ENTRY | EXPORT }]

[macro-label]	MC68851	
[macro-label]	MC68881	[fp-option,...]
[macro-label]	OPT	[{ALL | NONE | NOCLR}]
name	OPWORD	abs-expr
[macro-label]	ORG	[expr]
[macro-label]	PAGESIZE	[lines][,width]
[name]	PROC	[{ENTRY | EXPORT}]
[macro-label]	PRINT	[parameter],...
name	RECORD	[{ENTRY | EXPORT}] [, {INCR[EMENT] | DECR[EMENT] | MAIN}]
name	RECORD	{offset | IMPORT | {origin}} [, {INCR[EMENT] | DECR[EMENT]}]
name	REG	rlist
[macro-label]	SEG	[string]
name	SET	{*arith-expr* | *reg* | *import-name*}
[macro-label]	SPACE	[lines]

[macro-label]	STRING	{ ASIS | PASCAL | C }
[macro-label]	TITLE	str-expr
[macro-label]	WITH	name,. . .

Macro and SET variable directives

[macro-label]	ACTR	arith-expr
[macro-label]	AERROR	str-expr
[macro-label]	EXITM	
[macro-label]	{ GBLA | GBLC | LCLA | LCLC }	set-var-name , . . .
[macro-label]	GOTO	[±] { macro-label | str-exp}
[macro-label]	IF	bool-expr GOTO [±] { macro-label | str-exp}
[macro-label]	IF bool-expr THEN . . . assembler statements . . .	
	[ELSEIF bool-expr THEN . . . assembler statements . . .] . . .	
	[ELSE . . . assembler statements . . .]	
[macro-label]	ENDIF	

```
                    MACRO
[&name]             name[.&name]   [ &name [ : { INT | STR | A | C } ] [ { = | == } opnd-value ] ,... ]
                    - - -
                    macro definition
                    - - -
[macro-label]       ENDM

set-var-name        SETA           arith-expr
set-var-name        SETC           str-expr

[macro-label]       WHILE bool-expr DO
                    - - -
                    assembler statements
                    - - -
[macro-label]       ENDWHILE
[macro-label]       WRITE          write-item-list
[macro-label]       WRITELN        write-item-list
```

SET variable functions

&ABS(arith-expr)

&CHR(arith-expr)

&CONCAT(str-expr,...)

&DELSYMTBL(sym-tbl)

&DEFAULT(str-expr,str-expr)

&ENTERSYM(sym-tbl,symbol,value,flags)

&EVAL(str-expr)

&FINDSYM(sym-tbl,symbol)

&GETENV(str-expr)

{ &INTTOSTR | &I2S }(arith-expr,width,hex)

&ISINT(str-expr)

&LEN(str-expr)

&LEX(str-expr,start)

{ &LOWCASE | &LC }(str-expr)

&MAX(arith-expr, ...)

&MIN(arith-expr, ...)

&NBR(symb-param &SYSLIST)

&NEWSYMTBL

&ORD(str-expr)

&POS(str-expr,str-expr)

&SCANEQ(ch,str-expr,start)

&SCANNE(ch,str-expr,start)

&SETTING(str-exp)

str-exp ::=	value ::=
BLANKS	ON, OFF
BRANCH	SHORT, WORD, LONG
CASE	ON, OFF, OBJECT
CODEREFS	FORCEJT, NOFORCEJT,FORCEPC
DATAREFS	ABSOLUTE, RELATIVE
FORWARD	WORD, LONG
MACHINE	MC68000, MC68010, MC68020, MC68030
OPT	ALL, NONE, NOCLR
PRINT	ON, OFF, GEN, NOGEN, PAGE, NOPAGE, WARN, NOWARN, MCALL, NOMCALL, OBJ, NOOBJ, DATA, NODATA, MDIR, NOMDIR, HDR, NOHDR, LITS, NOLITS, STAT, NOSTAT, SYM, NOSYM
STRING	PASCAL, ASIS, C

{ &STRTOINT | &S2R }(str-expr)

{ &SUBSTR | &S }(str-expr,start,length)

&TRIM(str-expr, trim-left)

&TYPE(str-expr)

If str-expr is a macro variable name, then value ::=

UNDEFINED

PARM [STRUCTURED] { INT | STR }

{ SETA | SETC } '['ARRAY'['dim']]'

MACRO [{ FUNCTION | SYSVAR }]

if str-expr is a non-macro name, then value ::=

UNDEFINED

{ CODE | DATA } IMPORT

REG { An | Dn | ZAn | ZDn | CCR | SR | USP | MSP | SFC | DFC | CAAR | VBR | CACR | ISP | CRP | SRP | DRP | TC | PSR | PCSR | AC | CAL | SCC | VAL | BADn | BACn }

FPREG {FPn | FPCR | FPSR | FPIAR }

RLIST

FRLIST

FCRLIST

{ CODE | DATA } MODULE { EXPORT | ENTRY | IMPORT } [MAIN] ['(' type ')']

TEMPLATE [DATA IMPORT] ['(' type ')']

TEMPLATE FIELD ['(' type ')']

DATA FIELD [{ EXPORT | ENTRY | IMPORT }] ['(' type ')']

{ CODE | DATA } LABEL [{ EXPORT | ENTRY | IMPORT }] [MAIN] ['(' type ')']

SET

EQU

OPWORD

{ &UPCASE | &UC }(str-expr)

Appendix C

Assembly listing format

When the Macintosh Workshop Assembler produces a listing, it normally follows the format shown in Figure C-1. This format is based on the assumption that you will handle the listing in one of the following output forms:

- print it, using a monospace font
- edit it with an editor program that can scroll horizontally

The Assembler follows the format shown in Figure C-1 when all the listing-control default values are in effect; that is, when your source text includes only a TITLE directive and you specify no listing-control options in the Assembler command line. By using any other of the listing-control directives described in Chapter 4, or by using the listing-control Assembler options described in Appendix H, you can change the listing format.

Figure C-1
Default assembly listing format

```
MC68020 Assembler - Ver v.rr  <Title goes here>   dd-Mon-yy  Page xxx
Copyright Apple Computer, Inc. 1984, 1985, 1986

Loc   F Object Code     Addr  M Source Statement

xxxxx x xxxx xxxx xxxx  xxxxx x |-- machine instruction ------------>
              xxxx xxxx
              xxxx

xxxxx   xxxx xxxx xxxx        x |-- data statement ------------------>
        xxxx xxxx xxxx
               .
               .
               .
        xxxx xxxx xxxx
        xxxx xxxx x...
```

Figure C-1 shows the layout of an assembly listing, including a typical header plus a few listing lines. The *x*'s in Figure C-1 indicate the number of characters generated in each section; other italicized text indicates the type of information listed in that section. The six header lines (three of which are always blank) appear at the top of every page.

The first header line contains only the form feed character (ASCII $0C) that ejects each page.

The second header line contains the version and revision number of the Assembler, the title (if you wrote a TITLE directive in your source text), the date, and the page number. The Assembler formats this information according to these rules:

- The date and page number are right-justified on the page in conformance to the current PAGESIZE directive width setting.
- The title is truncated if it is too long for the available space.

The third header line always contains Apple's copyright notice.

The fourth and sixth header lines are always blank.

The fifth header line always contains the column headings for the listing. The *x*'s under the first five headings in Figure C-1 tell you the maximum number of characters that may appear in these columns. The six headings identify parts of each listing line, as described in the following paragraphs.

The Loc column identifies the location of the generated object code in the code or data module. This field is truncated to five hexadecimal digits or the number of digits you specified with an **-addersize** option when invoking the Assembler (see "Assembler Options" in Appendix H). It is blank if the corresponding source text does not generate any object code. There is no guarantee that this location is correct if an Assembler error has occurred. See Appendix D for a discussion of error reporting and recovery.

The F column contains generic and privileged instruction flags. If the Assembler converts a machine instruction from generic form, as described in Appendix A, the letter *G* appears in this column. If a machine instruction is privileged, the letter *P* appears. (Privileged instructions are described in the Motorola *M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual.*) In all other cases this column is blank.

The Object Code column contains the generated object code for the source line. This column is formatted one of three ways, depending on whether it contains a machine instruction, DC-generated data, or DCB-generated data:

- With machine instructions, all generated code is shown. The first line contains up to three words of four hexadecimal digits each. Each subsequent line, if any, contains up to two words, indented by one word.
- With data generated by a DC directive, every line contains up to three words. If you specified PRINT NODATA, then only the first line (or one line for each continuation) will appear. If you specified PRINT DATA, then up to 18 lines of data will appear. Any data remaining after 18 lines will be indicated by three periods, as illustrated at the bottom of Figure C-1.
- Data generated by a DCB directive appears in the same form as data generated by a DC directive; but instead of appearing only as words, it appears as bytes, words, or long words, depending on the directive's modifier, with a space between each text item. Such data listings are also limited to 18 lines.

The Object Code column also lists the values of defined symbols and SET variables. Register equates give the register name if the equate is to an A or D register; control registers are not shown. REG and FREG directive register-mask equates show a value consisting of two register masks, one in each direction, depending on the form of the MOVEM or FMOVEM instruction using the mask. EXPORT and ENTRY directives inside modules give the Loc-column locations corresponding to the specified labels.

The Addr column contains addresses accessed by machine instructions that have nonimported PC-relative address operands. It tells you the Loc-column value referenced by the instruction. As with the Loc column, the Addr-column value is truncated to five digits or to the number of digits specified by an **-addersize** Assembler option. All other addressing modes that access locations express offsets and so may be seen in the object code itself. If the instruction does not have a PC-relative address operand, or if its operand is a PC-relative reference to an imported identifier, this column is blank.

The M column lists the dynamic macro-nesting level, reduced modulo 10. If the listing line was not generated from a macro, the M column is blank.

The Source Statement column contains your source text line.

❖ *Note:* If you specified PRINT NOOBJ, either by a directive or by an Assembler option, then only the Loc and Source Statement columns will be displayed. This produces a more compressed listing with less information.

When the Assembler creates a listing file, it defines the creator as 'MPS ' and the type as TEXT. This lets you use the Macintosh Programmer's Workshop facilities to edit the listing. It also creates the following MPW Editor resource information to facilitate editing with the MPW Editor and printing with the MPW PRINT command:

- The listing's tabs are set to the same locations as the original source text tabs, or every four columns if the source text has no tabs. The Assembler places a tab in the listing file just before each entry in the source statement column. This gives the source statements in the listing the same format they had when being edited.
- The listing's font and font size are set according to the **-font** Assembler option. The standard default font is 7-point Monaco. However, a LaserWriter will adopt Courier as a default font when printing the listing.

Appendix D

Error and Warning Messages

The Assembler can report up to five errors or warnings for each source text line. It reports errors and warnings immediately after processing the line that caused them. If you are not making an assembly listing, the Assembler displays the source statement with the messages following it.

Error messages have the following format, designed to be compatible with the MPW Editor:

```
### Error nnn ### <error message>
    File "<filename>"; line xxx
    Macro "<name>"; line yyy
```

Warning messages have the same format, except that the word *Error* is replaced by the word "Warning." All warning messages may be suppressed by specifying PRINT NOWARN as a directive or as an Assembler option. Branch warnings alone may be suppressed by using the **–wb** Assembler option.

The first line shows the error number and the error message.

The second line shows the file in which the error occurred and the source line number in the file that caused the error. If the error occurred in a line of an included file, *filename* is the name of the included file. In either case, the filename and line number are formatted in such a way that you can select and enter them to execute the MPW File and Line commands. These commands open the file in a window and select the line that generated the error or warning.

The third line appears if the error occurs during macro expansion. In this case, the line number in line 2 refers to the outermost macro call, and the third line shows the macro name and the line number inside the macro that caused the error. The prototype statement is treated as line 1 of every macro. Depending on when the error is detected, the unexpanded macro line may be shown instead of the actual expansion.

There is no guarantee that error recovery is perfect; in a given line, additional errors may be reported that are caused by the occurrence of the first error. Also, correcting all the errors in a line may uncover further errors, since only five are reported.

- *Note:* Errors involving the syntax of machine instructions generally suppress the generated code. Therefore the location counter must be considered incorrect after any error in a machine instruction. Since location-counter values determine addresses and hence offsets in PC-relative effective addresses, a warning that a branch instruction can be changed (for example, that it can be made into a short branch) should be ignored if there are any errors between the branch and its destination. In general, it is advisable to ignore such warnings until all errors are removed.

All the errors and warnings reported by the Assembler are shown in Table D-1. In this table, the caret (^) indicates that the Assembler may insert additional information at that point. All messages with numbers lower than 200 are error messages, and all messages with numbers greater than 200 are warnings.

Table D-1
Error and warning messages

ID	Message
1	Invalid decimal constant
2	Invalid binary constant
3	Invalid octal constant
4	Invalid hex constant
5	Invalid string
6	Invalid macro label
7	Invalid label
8	Duplicate label
9	Invalid opcode
10	Operand stack overflow (expression too complex)
11	Operator stack overflow (expression too complex)
12	Id or integer constant expected in expression
13	Incomplete expression
14	Strings are only allowed in relations with other strings
15	Missing field name after ^
16	Undefined id: ^
17	Not a valid value: ^
18	Imported id not allowed: ^
19	Relocatable id not allowed: ^
20	Invalid arithmetic operation on relocatable id
21	Invalid id in expression: ^
22	Insufficient memory - unable to define ^
23	A label is required for this directive
24	A label is not allowed on this directive

25 An operand is not allowed for this directive
26 An operand is required for this directive
27 Temporary labels are not allowed on directives
28 Insufficient memory space to define string or floating-point constant
29 String does not represent a valid number
30 Invalid symbol in effective address
31 Cannot allocate space for literal
32 Invalid base register specification
33 ")" expected in effective address
34 Invalid register list
35 Address register or PC expected
36 Index register or PC expected
37 Index register expected
38 Illegal use of zero-suppressed register
39 Illegal size specification in effective address
40 <***Reserved for future use by ScanEA***>
41 Address register expected
42 "]" expected in effective address
43 Incomplete effective address
44 Outer-displacement expected
45 ")" or "," expected in effective address
46 Expression is not absolute
47 Strings used as expression operands must be 1 to 4 characters long
48 Invalid immediate expression

50 Illegal opcode or directive size attribute
51 Illegal characters in operand field
52 Illegal operand
53 Size mismatch for operator/operand
54 Illegal/missing operands or opcode not allowed for target machine
55 Invalid bit field offset or width
56 Invalid k-factor
57 Argument/subscript stack overflow (expression too complex)
58 Size attribute is not allowed on this directive
59 Temporary labels may only be used inside of code or data modules
60 Wrong number of arguments
61 Insufficient memory space for reference entry in Pass 2
62 Pass 2 system error involving references
63 Invalid operation on imports
64 Invalid constant type for operation
65 Operand out of range
66 Unpaired import may not be indexed
67 Invalid scale factor
68 Import not allowed in displacement
69 Scale factor not allowed on target machine
70 Invalid effective address for target machine

71 Long displacements are not allowed on target machine

75 No label permitted on directive
76 No operands permitted on directive
77 No operand size permitted on directive
78 Missing macro prototype line
79 Missing macro name
80 Missing ENDM directive on macro: ^
81 Macro variable undefined: ^
82 "*" dimension used for undefined dimension
83 Dimension exceeds max array dimension
84 Array dimension must be greater than 0
85 Type does not agree with original global definition: ^
86 Dimension does not agree with original global definition: ^
87 Out of memory for macro storage
88 Duplicate macro symbol definition: ^
89 INTERNAL ERROR--should not happen
90 Directive not allowed in macros: ^
91 Undefined macro or opcode: ^
92 Bad macro name: ^
93 Bad macro formal parameter name: ^
94 Macro label operand expected
95 Label not defined in current macro: ^
96 Statement not closed at end of macro: ^
97 Underflow of global loop counter
98 Underflow of loop counter for macro: ^
99 Directive cannot appear outside of macros: ^
100 Bad macro formal parameter (#^
101 Parameter not found: ^
102 Positional actual parameters not allowed after a keyword actual
103 Bad variable list, variable #^
104 Bad dimension specification for variable: ^
105 Bad variable name: ^
106 Unsubscripted reference to array: ^
107 Attempt to do integer assignment to string variable: ^
108 Bad macro variable subscript expression
109 Bad subscripted variable: ^
110 Subscript out of range for macro variable: ^
111 Subscripted reference to non-array variable: ^
112 Attempt to do string assignment to arithmetic variable: ^
113 Missing THEN or GOTO on IF directive
114 ELSE, ELSEIF, or ENDIF encountered without a corresponding IF
115 ENDWHILE directive expected
116 ENDIF directive expected
117 IF construct has more than one ELSE directive
118 ELSEIF not allowed after corresponding ELSE directive

119 Missing THEN on ELSEIF directive
120 Missing DO on WHILE directive
121 ENDWHILE with no corresponding WHILE
122 String value expected
123 Comma expected following WRITE or WRITELN list element
124 "[" expected following macro array variable: ^
125 Too many subscripts in macro array variable reference
126 Only one subscript is allowed to extract a character from a string variable
127 Substr or char extraction on non-string variable
128 &SYSLIST reference outside of a macro
129 &SYSLIST reference requires subscripting
130 &SYSLIST or macro parameter reference expected
131 A string with a length of one expected
132 Value must be in the range 0 to 255
133 Unable to read continuation line
134 Attempt to call a macro whose definition had errors: ^
135 Invalid use of string variable
136 Actual value for integer-typed macro parameter is not a number or too large: ^
137 Subrange value not in the range 0 to 255
138 Invalid or not supported &SETTING argument
139 Length argument to &SUBSTR cannot be negative
140 Backward GOTOs are not allowed outside of macros
141 Label not found in forward GOTO search of current macro: ^
142 Label not found in backward GOTO search of current macro: ^
143 Invalid or undefined &LIST macro string array variable
144 Value of an actual parameter exceeds 255 characters

150 Illegal use of CODE directive
151 Illegal use of DATA directive
152 Invalid register type
153 Illegal directive parameter
154 Relocatable expression is not allowed
155 No module to end
156 Missing operands or invalid character(s) in directive operand
157 Unable to open ^
158 Uncompleted conditional directive
159 Insufficient space for ^ to define qualification
160 Not allowed outside of module
161 ^ is already defined (possibly as different type)
162 Not allowed in data modules
163 Not allowed in RECORD templates
164 Only one MAIN code or data entry point is allowed in an assembly
165 Code may only be placed inside of code modules
166 No data module or template to end
167 PRINT stack overflow
168 PRINT stack underflow

169 Too many operands! (max of ^ allowed)
170 &Type requires a non-null string
171 Expansion of an &-reference is not allowed to cross lines
172 Unable to create DUMP file: ^
173 Unable to open LOAD file: ^
174 Write error during DUMP to ^
175 Not allowed inside of modules or templates
176 Not allowed inside of data modules or templates
177 Forward references to imports in data modules not allowed: ^
178 ^ is not one of the fields - origin not adjusted
179 Insufficient memory to adjust template origin
180 ^ is not a template or data RECORD name
181 No WITH to end
182 Location counter cannot change sign in modules or imported templates
183 ^ is not a reference to a template (type) name
184 Recursive field definitions are not allowed
185 Insufficient memory - unable to DUMP
186 Unable to define type for ^
187 Insufficient memory to evaluate recursive expression
188 Macro label not allowed on this directive
189 ^ is not a reference to a global template (type)
190 Invalid floating-point constant
191 Invalid ^ value
192 Duplicate ID not declared: ^
193 Reserved for future use
194 Unable to set the value for ^
195 Forward references to equates to imports (^) are not allowed

197 Reserved for future use!
198 ^
199 System Error
200 *Unused*
201 Argument to &StrToInt is not a number or too large: ^
202 Actual value for integer-typed macro parameter is not a number or too large: ^
203 String overflow (>255 characters) in &Concat
204 Odd location counter

210 A short branch could be used here
211 A word branch could be used here
212 Short branch used
213 Word branch used
214 Long branch used
215 Short branch to the next instruction changed to a NOP

219 END inside include file "^" terminates assembly
220 END statement missing
221 Extra character(s) in floating-point constant

222 Packed decimal with four or more exponent digits
223 Extra copies of long immediate generated -- one for each control register

230 Double long #immediate generated (use two DC's to define 16 hex digits)
231 Positive displacement
232 Odd displacement
233 Possible name conflict with global symbol: ^
234 Global definition of ^ used here

When the Assembler is opening, reading, or writing files, input/output (I/O) errors can occur. In addition to the standard Macintosh I/O error codes described in *Inside Macintosh*, the Assembler may return one of the error messages shown in Table D-2. In this table, errors marked with an asterisk (*) are the result of internal Assembler processes and should never be reported.

Table D-2
Assembler I/O errors

ID	Message
-31001	Not a text file
-31002	Not enough heap space to allocate I/O buffer
-31003	Not enough heap space to allocate I/O control block
-31004	I/O system error (illegal use of buffer) *
-31005	Too many includes *
-31006	Attempt to use I/O control block which is currently in use *
-31007	Illegal use of I/O control block *
-31008	Invalid file type
-31009	Inconsistent dump format

Appendix E

Other Assemblers

This appendix compares the MPW Assembler with other assemblers for the Motorola M68000 family and describes software that converts source text from some of the other assembly languages to the language described in this book.

Before the release of MPW, the primary development environments for the Macintosh were the Lisa Workshop (which includes the TLA Assembler) and the Macintosh 68000 Development System (MDS). The first half of this appendix compares MPW with these other two assemblers and with the Motorola assembler. The second half describes conversion programs that help you translate software written in TLA and MDS to the MPW Assembler environment.

Syntax comparison

This section compares the syntax accepted by the MPW Assembler with that accepted by the following other assemblers:

- The Motorola assembler (Mot)
- Apple® MDS
- Apple TLA

Writing identifiers

Table E-1 shows which characters may be used to compose identifiers in the four assembly languages, and how long those identifiers may be.

Table E-1
Identifier syntax rules

Rule	MPW	Mot	MDS	TLA
Lowercase letters (a..z)	yes[1]	yes[1]	yes[1]	yes[1]
Uppercase letters (A..Z)	yes[1]	yes[1]	yes[1]	yes[1]
Digits (0..9)	yes	yes	yes	yes
Underscores (_)	yes[1]	yes	yes[1]	yes[1]
Periods (.)	no	yes	yes[1]	yes
At symbols (@)	yes[2]	no	yes[2]	yes[2]
Dollar signs ($)	yes	no	yes	no
Number signs (#)	yes	no	no	no
Percent signs (%)	yes[1,3]	no	no	yes[1,3]
Maximum length	63	8	∞	8

1. Identifiers may start with these characters.
2. The at symbol is used to begin @-labels. It may be embedded inside identifiers with the MPW Assembler.
3. Leading percent signs should be avoided. They are reserved by Apple for special software such as the the Pascal runtime system. In the MPW Assembler identifiers beginning with % may not use 0 or 1 for a second character, because % is used to indicate binary numbers.

Writing numbers

Table E-2 shows how numbers are written in the four assembly languages. In this table, *d*... represents any sequence of integers.

Table E-2
Number syntax

Numerical base	MPW	Mot	MDS	TLA
Decimal	*d...*	*d...*	*d...*	*d...*
Hexadecimal	*$d...*	*$d...*	*$d...*	*$d...* or *d...H*
Binary	*%d...*	*%d...*	*%d...*	*d..B*
Octal		*@d...*	*^d...*	*d..O*
Floating-point	*"..."*[1]	*±d.d...* *:d...*		

1. Written as decimal or hexadecimal strings enclosed in quotation marks. Floating-point string formats may be any of those shown in the *Apple Numerics Manual*.

Writing strings

Table E-3 gives the rules and capabilities for writing strings in the four assembly languages. In this table, the ellipsis (...) indicates a sequence of characters.

Table E-3
String Syntax

Syntax rule	MPW	Mot	MDS	TLA
Form	'...'	'...'	'...'	'...' or "..."
Apostrophe representation	''	''	''	"...'..."
Ampersand representation	&&[1]	&	&	&
Generates Pascal strings	yes	no	yes	no
Generates C strings	yes	no	yes	no

1. In macros only.

Defining modules

Table E-4 compares the way that the four assembly languages define code and data modules in source text.

Table E-4
Module definition

Module	MPW	Mot	MDS	TLA
Code	PROC...ENDP[1]	control section	one module[2]	PROC[3]
Data	RECORD...ENDR[1]	control section	DS directive[4]	none

1. Modules may be local to a file.
2. MODULE directive permits multiple code modules in MDS 2.0
3. Modules are always exported.
4. Only DS directives generate A5-relative data.

Communicating between modules

Table E-5 compares the directives that allow each assembly language to transfer references to the source text between modules.

Table E-5
Communication directives

MPW	Mot	MDS	TLA
EXPORT	XDEF	XDEF	DEF
IMPORT	XREF	XREF	REF, REFA5, REF32
ENTRY			

Writing expressions

Table E-6 lists the operators you can use when writing expressions in each of the four assembly languages. Some MPW operators may be written in more than one way. All alternatives are shown.

Table E-6
Allowable operators

Operation	MPW[1]			Mot[1]	MDS[1]	TLA[2]
addition	+			+	+	+
subtraction	–			–	–	–
multiplication	*			*	*	*
division	/	DIV	÷	/	/	/
modulus reduction	//	MOD			\	
logical or	++	OR	¦	!	!	¦
exclusive-or	--	XOR			^	
logical and	**	AND		&	&	
equal	=					=
not equal	<>	≠			<>	
less than	<					
greater than	>					
less than or equal	<=	≤				
greater than or equal	>=	≥				
shift left	<<			<<	<<	
shift right	>>			>>	>>	
one's complement	~					~
negation	–			–	–	–
logical not	¬	NOT				

1. Operators have precedence; parentheses are allowed.
2. No operator precedence; no parentheses allowed. Angle brackets (greater than and less than symbols) are used in place of parentheses.

Location-counter reference

The meaning of the location-counter symbol (*) varies between the four assembly languages when used in the DC directive (WORD or LONG in TLA). For example, in the statement

```
label       DC.W        1,2,*-X,3,4
```

the value of * is defined as the value of the label—the location of the first word—in the MPW, Motorola, and TLA assemblers. In MDS, the value of * is the location of the word representing the current operand. Therefore in MDS, * represents the location of the third word.

Addressing features

The four assembly languages include different features for writing addresses, as shown in Table E-7.

Table E-7
Addressing features

Feature	MPW	Mot	MDS	TLA
MC68020 addressing	yes	yes	no	no
Bases	A5 default	A5 for data	A5 for DS	no
Qualified identifiers	yes	no	no	no
Data structures	templates	OFFSET, EQU	EQU	EQU

Writing macros

The MPW Assembler Macro Processor does not accept macro definitions written in any of the other three languages. In all four assemblers, however, macro calls have the same basic form.

The MPW Assembler's macro processor supports all the features of the other assemblers with one exception: macros that generate only part of a statement (such as only an operand) are not supported. Such macros can be written only in the MDS Assembler. The MPW Assembler also supports a number of features not found in any of the other assemblers, such as keyword macros and SET variables.

Hence you can always rewrite Motorola, TLA, and MDS macro definitions into MPW form, except for MDS macros that generate partial source lines. In most cases, you can leave macro call directives as they were originally written.

Conversion programs

The following sections describe in detail the operations performed by programs that convert TLA and MDS source texts into a form suitable for the MPW Assembler. These programs are provided as part of the MPW software. Instructions for using them are given in the *Macintosh Programmer's Workshop Reference.*

TLA conversion

TLACvt is a Macintosh program that you can use to translate Lisa Workshop TLA assembler source files to the syntax required by the MPW Assembler. TLACvt translates the following source text elements:

- tokens within statements and directives
- special tokens within macros
- directives

The conversions that TLACvt performs are listed in Tables E-8 through E-10.

Table E-8
Normal TLA token conversion

TLA	MPW	Notes
<	(	Used only in expressions[1]
>	)	Used only in expressions[1]
\	MOD	Used only in expressions
&	AND	Used only in expressions
^	XOR	Used only in expressions
id.id	*id_id*	Periods not allowed in MPW identifiers
"..."	'...'	Embedded single quotation marks are doubled
*dddd*H	$*dddd*	The H is removed and the value preceded with $
*dddd*B	%*dddd*	The B is removed and the value preceded with %
*dddd*O	$*dddddddd*	Octal is converted to eight-digit hex numbers

1. Because TLA has no operator precedence when evaluating expressions, the converted expressions, while syntactically correct, may evaluate to different values in MPW. For example, 2+3*4 evaluates to 20 in TLA and to 14 in the MPW Assembler. TLA expressions containing more than one operator should be carefully checked.

Table E-9
TLA macro token conversions

TLA	MPW	Notes
%*n*	&EVAL(&SYSLIST[*n*])	Done only in IF directives[1]
'%*n*'	&SYSLIST[*n*]	Done only in IF directives[1]
"%*n*"	&SYSLIST[*n*]	Done only in IF directives[1]
%*n*	&SYSLIST[*n*]	Throughout macros except IF directives[1]
'%*n*'	'&SYSLIST[*n*]'	Throughout macros except IF directives[1]
"%*n*"	'&SYSLIST[*n*]'	Throughout macros except IF directives[1]
;	.*	Macro comments only[2]

1. You should check these macro conversions carefully. The conversions shown are somewhat generic and may not convey the intent of the original program. We recommend that you convert your macros by hand, using explicit formal macro parameters rather than &SYSLIST.
2. TLA comments starting with a semicolon at the beginning of a line remain unchanged when used outside macros. Inside macros, the semicolon is converted to a period-asterisk (.*).

Table E-10
TLA Directive Conversions

TLA			MPW			Notes
	.MAIN	id	id	MAIN	EXPORT	All code names are .DEF
	.PROC	id	id	PROC	EXPORT	All code names are .DEF
	.FUNC	id	id	FUNC	EXPORT	All code names are .DEF
	.DEF	id,...		EXPORT	id,...	
	.REF	id,...		IMPORT	id,...	
	.REFA5	id		IMPORT	id:DATA	
	.SEG	'name'		SEG	'name'	Null or 8 blanks becomes 'Main'
	.END			END		
	.ASCII	'string'		DC.B	'string'	
	.BLOCK	l		DCB.B	l,0	Special case for .BLOCK
	.BLOCK	l,e		DCB.B	l,e	For DC's and DCB's ...
	.BYTE	e,...		DC.B	e,...	... if in macro definition, or
	.WORD	e,...		DC.W	e,...	if e contains a string, the e
	.LONG	e,...		DC.L	e,...	is enclosed in parentheses.
	.ALIGN	e		ALIGN	e	ALIGN 0 cancels auto alignment
	.ORG	e		ORG	e	
	.RORG	e		ORG	e	ORG and RORG same in TLA
id	.EQU	e	id	EQU	e	
	.MACRO	name		MACRO name		There is no guarantee that macros as converted will work in the MPW Assembler
	- - -			- - -		
	.ENDM			ENDM		
	.IF	e		IF	e THEN	
	.ELSE			ELSE		
	.ENDC			ENDIF		
	.LIST			PRINT	ON	
	.NOLIST			PRINT	OFF	
	.PAGE			EJECT		
	.TITLE	'string'		TITLE	'string'	
	.MACROLIST			PRINT	GEN	
	.NOMACROLIST			PRINT	NOGEN	
	.INCLUDE	name		INCLUDE	'name'	Name should be changed
	.INCLUDE	name.SYM		LOAD	'name.SYM'	Name should be changed
	.DUMP	name		DUMP	'name'	Name should be changed
	.PATCHLIST		;?	.PATCHLIST		Commented out
	.NOPATCHLIST		;?	.NOPATCHLIST		Commented out
	.OUT	'string'	;?	.OUT	'string'	Commented out
	.REF32	name	;?	.REF32	name	Commented out
	id			id#		Done only if id op is the same as an MPW directive

Limitations on TLA conversion

The following paragraphs list some of the problems you may encounter in performing a TLA-to-MPW conversion with TLACvt:

- TLA operators all have the same precedence. Hence TLA expressions may be evaluated differently after they are converted to MPW, where they are subject to the rules given under "Evaluation of Expressions" in Chapter 2.
- In TLA, you can write macro calls and directives in the first field of the source text line, which becomes the label field when the source text is converted to MPW. When TLACvt strips the initial period from directives in the first field, MPW treats them as labels or macro calls.
- TLA treats effective addresses of the form *label*(A*n*), where *label* is a code module label, as mode 73. After conversion, MPW interprets this form as mode 5. You can force MPW to interpret such addresses correctly by changing their TLA form to *label*(PC,A*n*), *label*(A*n*.W), *label*(A*n**1), or *label*(A*n*.W*1). Effective address modes are shown in Table 3-2.

Using TLACvt

To use TLACvt, send a command with the following syntax to the MPW Shell:

TLACvt [*option*]... [*file*]...

If no *file* is specified, standard input in converted. If the input file name is *name*, the converted output is written to *name*.a. If input is from the standard input file, the converted output is written to standard output. You can use the **-n**, **-prefix**, and **-suffix** options to modify the output file naming conventions.

Each *option* is preceded by a hyphen. The TLACvt options are the following:

-d	Detab the input. All tabs are removed and replaced with spaces. The number of spaces is determined by the tab setting. (See the **-t** option below.)
-e	Detab the input (as done by the **-d** option) and entab the output as a function of the tab setting. (See the **-t** option below.)
-f *directivesFile*	The casing of directives is controlled by the file of directives specified by *directivesFile*. The file TLACvt.Directives is supplied for this purpose; you can edit it to change the capitalization. By default, all directives are converted to uppercase.
-m	Do *not* insert TLA-compatible mode-setting directives (BLANKS ON and STRING ASIS) into converted source.

-n	Do not add the .a extension to the input filename to produce the output filename. If you specify this option, you must also specify **-prefix** or **-suffix.**
-p	Writes TLACvt's version information and conversion status to diagnostic output.
-pre[fix] *string*	If the input file name is Name, the output filename is produced by prefixing *string* to the name, that is, "*string*Name.a". (The ".a" suffix may be suppressed by using the **-n** option or changed by using the **-suffix** option.)
-suf[fix] *string*	If the input file name is Name, the output filename is produced by appending *string* to the file name, that is, "Name*string*". The default suffix is ".a".
-t *value*	Set the output file's tab value to *value* (2 to 255). The default is to use the input file's tab setting (if there is one); otherwise a value of 8 is assumed. (8 is the default used by the Lisa Workshop's MacCom utility when transferring text files—it's assumed that MacCom was used to transfer the TLA files from the Lisa to the Macintosh.)
-u *c*	When TLACvt detects a name in the opcode field that is the same as an MPW directive, it appends the character *c* to make the name unique. (The default character is **#**.)

MDS conversion

MDSCvt is a Macintosh program that you can use to translate MDS assembler source files to the syntax required by the MPW Assembler. MDSCvt translates the following source text elements:

- tokens within directives and directives
- special tokens within macros
- directives

The conversions that MDSCvt performs are listed in Tables E-11 through E-13.

Table E-11
Normal MDS token conversion

MDS	MPW	Notes
&	AND	Used only in expressions
!	OR	Used only in expressions
id.id	*id_id*	Periods not allowed in MPW Assembler ID's
^*dddd*	$*dddddddd*	Octal is converted to eight-digit hexadecimal numbers

Note: All other arithmetic operators and all operator precedence recognized by the MDS Assembler are compatible with the MPW Assembler.

Table E-12
MDS macro token conversions

MDS	MPW	Notes
%*n*	&EVAL(&SYSLIST[*n*])	Done only in IF directives in a TLA macro[1]
'%*n*'	&SYSLIST[*n*]	Done only in IF directives in a TLA macro[1]
"%*n*"	&SYSLIST[*n*]	Done only in IF directives in a TLA macro[1]
%*n*	&SYSLIST[*n*]	Throughout TLA macro except IF directives[1]
'%*n*'	'&SYSLIST[*n*]'	Throughout TLA macro except IF directives[1]
"%*n*"	'&SYSLIST[*n*]'	Throughout TLA macro except IF directives[1]
{*id*}	&EVAL(&*id*)	Done only in IF directives in a MDS macro[1]
{*id* | *n*: *m*}	&EVAL(&*id*[*n*,*m*])	Done only in IF directives in a MDS macro[1]
'{*id*}'	&*id*	Done only in IF directives in a MDS macro[1]
'{*id* | *n*: *m*}'	&*id*[*n*,*m*]	Done only in IF directives in a MDS macro[1]
{*id*}	&*id*	Throughout MDS macros except IF directives[1]
{*id* | *n*: *m*}	&*id*[*n*,*m*]	Throughout MDS macros except IF directives[1]
'{*id*}'	'&*id*'	Throughout MDS macros except IF directives[1]
'{*id* | *n*: *m*}'	'&*id*[*n*,*m*]'	Throughout MDS macros except IF directives[1]
;	.*	Macro comments only[2]

1. The MDS Assembler supports two forms of macro definition: one unique to MDS and the other more-or-less compatible with TLA. The converter attempts to convert both forms. Because MDS allows partial-line macros and macro calls from all assembler fields, some conversions are not possible; see below, "Limitations on MDS Conversion." You should check these macro conversions carefully. The conversions shown are somewhat generic and may not convey the intent of the original program. We recommend that you convert your macros by hand, using explicit formal macro parameters rather than `&SYSLIST`. Partial-line MDS macros will have to be completely rewritten.
2. MDS comments starting with a semicolon at the beginning of a line remain unchanged when used outside macros. Inside macros, the semicolon is converted to a period-asterisk (.*).

Table E-13
MDS directive conversions

MDS			MPW			Notes
	XDEF	id,...		EXPORT	id,...	
	XREF	id,...		IMPORT	id,...	
	BSR	id		JSR	id	Only if id is XREFed
	DC.size	e		DC.size	e	
	DS.size	e,...		DS.size	e,...	
	DCB.size	l,e		DCB.size	l,e	
	ALIGN	e		ALIGN	e	
id	EQU	e	id	EQU	e	
id	SET	e	id	SET	e	
id	REG	rlist	id	REG	rlist	
	.MACRO	name		MACRO		TLA macro syntax;
				name		there is no guarantee that
	- - -			- - -		macros as converted will
	.ENDM			ENDM		work in the MPW Assembler
	MACRO name [p_1,...] =			MACRO		MDS macro syntax;
				name	[&p_1,...]	partial-line macros cannot be
	- - -			- - -		converted. Nothing can follow
				ENDM		the "=" on the MACRO line.
	IF	e		IF	e THEN	
	ELSE			ELSE		
	ENDIF			ENDIF		
	.NOLIST			PRINT	OFF	
	EJECT			EJECT		

INCLUDE name	INCLUDE 'name'	Name should be changed
INCLUDE name.D	LOAD 'name.D'	LOAD if .D suffix
.DUMP name	DUMP 'name'	Name should be changed
RESOURCE ... ;?	RESOURCE ...	Commented out
.LISTTOFILE ;?	.LISTTOFILE	Commented out
.LISTTODISP ;?	.LISTTODISP	Commented out
.VERBOSE ;?	.VERBOSE	Commented out
.NOVERBOSE ;?	.NOVERBOSE	Commented out
.TRAP name $Axxx name trapname,x [opnd]	OPWORD $Axxx trapname [opnd],x	Example: _READ,Async Becomes: _READ Async
STRING_FORMAT e	STRING C STRING PASCAL STRING ASIS	Dependent on the current STRING_FORMAT settings (bits 0 and 1).
END	END	
id	id#	Done only if id op is the same as an MPW directive

MDS conversion strategy

An MDS assembly usually represents a single code module, plus possibly a data module generated from DS directives (although MDS 2.0 allows multiple code modules). All the DS directives from all MDS modules linked by the MDS linker in a single assembly are assigned descending (negative) addresses relative to A5, starting at the address-$100(A5). The $100 bytes are reserved for QuickDraw's globals, since all MDS main programs start by calling InitGraf with the nonrelocatable address -4(A5) on the stack. The value $100 value can be changed by specifying the **-g** option with MDSCvt.

In order to convert MDS files to something equivalent in the MPW Assembler, MDSCvt tries to create a source text structure as close as possible to the following skeleton outline:

```
[PROC       MAIN]          ; Placed in front of the module
BLANKS      ON             ; Blanks are to be ignored
STRING      ASIS           ; Can be changed by STRING_FORMAT

[ENTRY      id:MAIN]       ; Define "id" as main program entry point
[DATA       MAIN           ; The ORG -$100 is done only if
ORG         -$100          ;     this is a MAIN module
CODE ]
 - - -      ; Code
DATA        DECR           ; First DS encountered in file
DS          e              ; Any number of DS's
 - - -
CODE        ; After the DS's
 - - -      ; More code
DATA        ; More DS's
DS          e
- - -
CODE        ; Back to code section
- - -      ; More code
[END]       ; End of module
```

An MDS file corresponds to only one code module. If it is not an included file, MDSCvt tries to declare the entire module as a single MPW code module by placing a PROC or MAIN directive at the start of the file. The code module must be declared as a MAIN module if the file represents the main program. If the main program's entry point is not at the start of the module, an ENTRY statement is generated to specify the identifier in a **-!** MDSCvt option as the main entry point. When converting an included file, MDSCvt does not generate PROC, MAIN, or END.

MDS 2.0 permits multiple code modules, causing MDSCvt to generate additional PROC statements. In MDS, all labels defined by EQU and SET have global scope; in MPW, they have local scope. You may need to adjust the scope of labels after the conversion of an MDS program with more than one code module.

DS directives are placed in a local decrementing data module (between DATA and CODE directives). This is equivalent to the decrementing allocation done by the MDS Linker. The MPW Linker will collect the data modules and place them below A5 just as the MDS Linker does. But the MPW linker does not have provision for reserving the $100 bytes for QuickDraw; hence the space must be reserved explicitly. The data module that will contain this space must be declared as a MAIN data module so that it is loaded immediately below A5 in order to make the -4(A5) address work for the InitGraf call. To make all this happen, MDSCvt must know that it is converting a main module. You indicate this by specifying an option (**-main**, **-l**, or **-g**) in the command line when MDSCvt is executed, as explained below under "Using MDSCvt."

Limitations on MDS Conversion

The following list describes some of the problems you may encounter in performing an MDS-to-MPW conversion with MDSCvt:

- The tabs in the input file may affect the spacing of the output.
- The STRING_FORMAT directive may be converted incorrectly if strings are generated from macros.
- Since the STRING_FORMAT directive controls the format of strings generated by DC directives and PEA and LEA literals independently, string formats are ignored in macro definitions. This is because MDSCvt processes STRING_FORMAT to determine when to generate the MPW STRING directive.
- Partial-line macros cannot be converted, because MPW only allows macros invoked from the opcode field. The macro definition is flagged with a ;? sequence, but the macro call cannot be detected.
- EQU's to effective addresses or undefined symbols are not supported by the MPW Assembler and go undetected in the conversion.
- EQU's to strings are not supported by the MPW Assembler and go undetected in the conversion. All MPW equates are numeric. MPW interprets an MDS directive of the form x EQU '*' as a directive to set the value of *x* equivalent to the ASCII value of *, and not to the string defined by the MDS current string format.
- Using the location counter (*) in DC directives has a different meaning in MDS than that defined in the MPW Assembler. In MDS, the * symbol in DC directives is whatever value it has at the time the * is encountered in the directive. In the MPW Assembler, the * corresponds to the first byte of the directive, after alignment.
- Indented labels without a delimiting colon are not detected by the converter.

- In MDS, both opcodes and directives may be placed at the start of an input line. This is allowed because labels cannot be the same as opcode or directive names. Thus an opcode or directive in the first character position can be uniquely detected by the MDS assembler. This is not the case with the MPW Assembler. Opcodes and directives may not be placed in the first character position and labels can be the same as opcode or directive names. If the converter encounters an MDS directive in the first character position, it will prefix it with a blank and convert it as appropriate. However, the converter does not detect opcodes in the first character position; it treats them as labels.

Using MDSCvt

To use MDSCvt, send a command with the following syntax to the MPW Shell:

MDSCvt [*option*] ... [*file*]...

Standard input is converted if no *files* are specified. If the **-main** option is used, only one file should be specified. If input is from the standard input file, the converted output is written to standard output. If the input filename is *Name*, the converted output is written to *name*.a. The **-n**, **-prefix**, and **-suffix** options let you modify the output file naming conventions.

Each *option* is preceded by a hyphen. The MDSCvt options are the following:

-d Detab the input. All tabs are removed and replaced with spaces. The default setting is 8 spaces; this value can be changed with the **-t** option.

-e Detab the input (like the **-d** option), and entab the output as a function of the tab setting (either 8, or the value specified with the **-t** option).

-f *directivesFile* Set the case (upper/lower) of directives according to the entries in *directivesFile*. The file MDSCvt.Directives is supplied for this purpose; you can edit it to change the capitalization. If you don't use this option, then all directives are converted to uppercase.

-g *globals* Convert a main program source and reserve globals space below A5. *Globals* may be specified in decimal or hexadecimal (by preceding the value with a $). The value specified must be negative. For example,

```
-g -512
-g $200
```

-i Convert include files. No PROC or MAIN or END will be generated by MDSCvt.

-m	Do *not* insert MDS-compatible mode-setting directives (BLANKS ON and STRING ASIS) into the converted source.
-main	Convert a main program source. The conversion is done to make the file look like the main code and data modules. Only one file should be converted when using this option.
-n	Do not add the ".a" extension to the input filename to produce the output filename. If you use **-n**, you must also specify either **-prefix** or **-suffix**.)
-p	Write MDSCvt's version information and conversion status to the diagnostic output file.
-pre[**fix**] *string*	If the input filename is "Name", the output filename is produced by prefixing the *string* to Name, that is, "*string*Name.a". (You can suppress the ".a" suffix by using the **-n** option, or change it by using the **-suffix** option.)
-suf[**fix**] *string*	If the input filename is "Name", the output filename is produced by appending the *string* to the filename, that is, "Name*string*". The default suffix is .a.
-t *value*	Set the tab setting for input and output files to the specified value (2 to 255). The default setting is assumed to be 8.
-u *c*	When MDSCvt detects a name in the opcode field that is the same as an MPW directive, it appends the character *c* to make the name unique. (The default character is **#**.)
-! *identifier*	Convert a main program source and define the main program's entry point by the specified identifier. This option corresponds to the MDS linker's **!** command.

Appendix F

Macintosh Character Set

Second digit \ First digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	space	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“		
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”		
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈	’		
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷		
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	Σ	«	◊		
8	BS	CAN	(	8	H	X	h	x	à	ò	®	Π	»	ÿ		
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…			
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	␣			
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª	À			
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º	Ã			
D	CR	GS	–	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ			
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ			

␣ stands for a nonbreaking space, the same length and width as a digit.

Shaded characters cannot normally be generated from the Macintosh keyboard or keypad.

Shaded characters are not in all fonts.

Codes $D9 through $FF are reserved for future expansion.

Appendix G

Instruction Sets

This appendix defines the instruction sets accepted by the MPW. They are equivalent to the MC68000, MC68010, MC68020, MC68881, and MC68851 instruction sets described in more detail in the Motorola *M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual,* the Motorola *MC68881 Floating-Point Coprocessor User's Manual,* and the Motorola *MC68851 Paged Memory Management Unit User's Manual.* Refer to those manuals for full descriptions of these instructions.

❖ *Note:* Some mnemonics have been changed to eliminate ambiguities or to conform to the Motorola Assembler forms. If in doubt, check your mnemonics with those given in Tables G-7, G-8, and G-9.

The Macintosh instruction sets contain certain machine instructions that encode into more than one bit configuration, depending on the instruction's operands. Each instruction consists of an opcode word and zero or more extension words. The opcode word contains some basic constant information about the instruction, but other fields must be set to indicate the kinds of operands (effective addresses) and the size of the instruction.

Instruction evaluation

The Assembler determines the encoding for an instruction by looking at the group corresponding to the mnemonic. Starting with the first encoding line in the group, the Assembler checks the machine type, the size, the source operand mode, the destination operand mode, and (where applicable) the immediate data range. If all the information matches, the Assembler generates the code for the instruction, through its encoding group number. If any one of the items doesn't match, the next encoding (if any) in the group is checked. This process continues until an encoding is found or the end of the group is reached.

If the Assembler reaches the end of the group before finding a valid encoding (including coprocessor opcodes), it indicates an invalid instruction. It then tries to interpret the source text line as a macro. Finally, it tries to interpret it as an OPWORD directive.

Listing conventions

Tables G-2, G-6, G-7, G-8, and G-9 list the instructions and condition codes accepted by the MPW Assembler. Table G-7, covering the processor instructions for the MC68000, 68010, and 68020, is divided into seven columns with the following headings:

- Opcode: The mnemonic you write in your source text.
- Operands: The operands (if any) required by the opcode.
- Opcode word: The binary encoding of the first word or extension word of the instruction.
- Group: The encoding group number assigned to the instruction.
- Flags: Letters indicating specific characteristics of the instruction.
- Range: A code number identifying the instruction's data range.
- Equivalent: The actual code equivalent for generic instructions.

Tables G-8 and G-9, covering the coprocessor instructions, have a slightly different set of column headings:

- Opcode: The mnemonic you write in your source text.
- Operands: The operands (if any) required by the opcode.
- Opcode word: The binary encoding of the first word of the instruction.
- cp-type. The coprocessor instruction type.
- Group: The encoding group number assigned to the instruction.
- Flags: Letters indicating specific characteristics of the instruction.

- Equivalent: The actual code equivalent for generic instructions.

Each column is described in more detail in the paragraphs below.

Opcode

This column contains the legal instruction mnemonics recognized by the Assembler. For further information about the instructions they represent, see the appropriate Motorola manual listed at the beginning of this appendix.

Operands

This column may contain register names (such as D*n* or A*n*) or nonterminal symbols (such as ea*n* or R*c*). The nonterminal symbols stand for addressing modes. Table G-1 shows all possible operand forms. A figure *1* in the table indicates that the corresponding addressing mode is legal. A figure *0* means it is illegal. Addressing modes are further described in the Motorola manuals and in Chapter 3 of this manual.

Table G-1
Instruction Operands

Operand	ea0	ea1	ea2	ea3	ea4	ea5	ea6	ea7	ea8	ea9	ea10	ea11	ea12	Rc
Special	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D*n*	1	1	0	1	0	0	1	0	0	1	1	0	0	0
A*n*	1	1	0	0	0	0	0	0	0	0	0	0	0	0
(A*n*)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
(A*n*)+	1	1	1	1	0	1	1	0	1	0	0	0	1	0
-(A*n*)	1	1	1	1	1	1	1	0	0	0	0	0	1	0
d(A*n*)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
d(A*n*,X*n*)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
(bd,A*n*,X*n*)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
([bd,A*n*,X*n*],*od*)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
([bd,A*n*],X*n*,*od*)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
(*ae*).W | (*ae*).L	1	1	1	1	1	1	1	1	1	1	1	1	1	0
#Data	1	0	0	0	0	0	1	0	0	0	0	0	1	0
Label	1	0	0	0	0	1	1	1	1	0	1	0	1	0
d(PC)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
d(PC,X*n*)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
(*bd*,PC,X*n*)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
([*bd*,PC,X*n*],*od*)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
([*bd*,PC],X*n*,*od*)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
RList	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CCR	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SR	0	0	0	0	0	0	0	0	0	0	0	0	0	0
USP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Ctl Regs	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Rc ::= SFC | DFC | CACR | VBR | CAAR | MSP | ISP

Opcode word

This column contains the binary encoding of the fixed information placed in the first word of the instruction corresponding to the specified mnemonic.

cp-type

This column lists the instruction type for coprocessor instructions. The meaning of the type codes in this column are shown below.

Type	Meaning
Gen1	General coprocessor instruction, register operand(s) or no operand
Gen2	General coprocessor instruction, memory to register(s)
Gen3	General coprocessor instruction, register(s) to memory
Bcc	Branch on coprocessor condition
DBcc	Decrement and branch on coprocessor condition
Rest	Coprocessor restore instruction (privileged)
Save	Coprocessor save instruction (privileged)
Scc	Set on coprocessor condition
Tcc1	Trap on coprocessor condition, predicate supplied by processor
Tcc2	Trap on coprocessor condition, operand predicate

Group

This column lists the Assembler's encoding group for that mnemonic. The set of all MPW Assembler instructions may be viewed as a collection of subsets, with each subset corresponding to a specific encoding. There are 49 distinct encoding groups, numbered 0 to 48.

Flags

This column indicates various attributes about the instruction. The meanings of the flag symbols are given in below.

Flag	Meaning
1	MC68010 instruction
2	MC68020 instruction
P	Privileged instruction
G	Generic instruction
B	Byte data size accepted
W	Word data size accepted
L	Long word data size accepted
S	Single precision data size accepted
D	Double precision data size accepted
X	Extended data size accepted
K	Packed BCD data size accepted

Range

This column specifies the legal range for absolute data, if the range is not otherwise expressed. The codes that specify the ranges are described below.

Number	Meaning
0	Value=0
3	Value ::= 1..8; 0 indicates a value of 8
4	4-bit unsigned value
8	8-bit unsigned value
-8	8-bit signed value
16	16-bit unsigned value
-16	16-bit signed value
32	32-bit unsigned value

Equivalent

This column indicates what actual instruction the mnemonic represents in cases where the instruction form is generic. Generic instructions are described in Appendix A.

Condition codes

All combinations of instructions and condition codes that the MPW Assembler accepts are listed in Tables G-3, G-4, and G-5. The condition codes alone are listed in Table G-2.

Table G-2
MC68xxx condition codes

Mnemonic	Condition	Encoding	Test
T	True	0000	1
HI	High	0010	$\overline{C} \cdot \overline{Z}$
LS	Low or same	0011	C+Z
CC,CS	Carry clear	0100	$\overline{C}$
CS,LO	carry set	0101	C
ME,NZ	Not equal	0110	$\overline{Z}$
EQ,Z	Equal	0111	Z
VC	Overflow clear	1000	$\overline{V}$
VS	Overflow set	1001	V
PL	plus	0101	$\overline{N}$
MI	Minus	1011	N
GE	Greater than or equal to	1100	$N \cdot V + \overline{N} \cdot \overline{V}$
LT	Less than	1101	$N \cdot \overline{V} + \overline{N} \cdot V$
GT	Greater than	1110	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
LE	Less than or Equal to	1111	$Z + N \cdot \overline{V} + \overline{N} \cdot V$

Table G-3
MC68881 IEEE non aware tests

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not equal	$\overline{Z}$	001110
GT	Greater than	$\overline{NAN+Z+N}$	010010
NGT	Not greater than	NAN+Z+N	011101
GE	Greater thanor equal to	$Z+\overline{(NAN+N)}$	010011
NGE	Not (greater than or equal to)	NAN+(N•Z)	011100
LT	Less than	$N•\overline{(NAN+Z)}$	010100
NLT	Not less than	NAN+Z+N	011011
LE	Less than or equal to	$Z+\overline{(N+NAN)}$	010101
NLE	Not (less than or equal to)	NAN+$\overline{(N+Z)}$	011010
GL	Greater than or less than	$\overline{NAN+Z}$	010110
NGL	Not greater than or less than	NAN+Z	011001
GLE	Greater than , less than, or equal to	$\overline{NAN}$	010111
NGLE	Not (greater than, less than, or equal to)	NAN	011000

Table G-4
MC68881 IEEE aware test

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not equal	Z	001110
OGT	Orderd greatrer than	NAN+Z+N	000010
ULE	Unordered less than or equal to	NAN+Z+N	001101
OGE	Ordered greater than or equal	Z+(NAN+N)	000010
ULT	Unordered less than	NAN+(N•Z)	000010
OLT	Ordered less than	N•(NAN+Z)	000010
UGE	Unordered or greater than	NAN+Z+N	000010
OLE	Ordered less than or equal to	Z+(N+NAN)	000010
UGT	Unordered or greater than	NAN+(N+Z)	000010

OGL	Ordered greater than or less than	NAN+Z	000010
UEQ	Unordered or equal to	NAN+Z	000010
OR	Ordered	NAN	000010
UN	Unordered	NAN	000010

Table G-5
MC68881 Miscellaneous tests

Mnemonic	Definition	Equation	Predicate
F	False	False	000000
T	True	True	001111
SF	Signalling false	False	010000
ST	Signalling true	True	011111
SEQ	Signalling equal	Z	010001
SNE	Signalling not equal	Z	011110

Table G-6
MC68851 PMMU condition codes

Mnemonic	Condition	Encoding
BS	B set	000000
BC	B clear	000001
LS	L set	000010
LC	L clear	000011
SS	S set	000100
SC	S clear	000101
AS	A set	000110
AC	A clear	000111
WS	W set	001000
WC	W clear	001001
IS	I set	001010
IC	I clear	001011
GS	G set	001100
GC	G clear	001101

CS	C set	001110
CC	C clear	001111

Instruction set listings

Tables G-7, G-8, and G-9 are edited listings of the actual data files used to produce the opcode table used by the Assembler. These tables show all the opcodes sorted alphabetically. Different encodings for the same mnemonic are grouped, with a blank line separating each group. The encodings within each group are ordered so that generic forms or optimizations occur before more general forms.

In Tables G-8 and G-9, the following metasymbols are used to denote groups of coprocessor registers:

FPn	::=	FP0..FP7
<FRList>	::=	Floating-point register list
<FCRList>	::=	Floating-point control register list
BADn	::=	BAD0..BAD7
BACn	::=	BAC0..BAC7
xRP	::=	CRP | SRP | DRP
SCCCAL	::=	SCC | CAL

Table G-7
MC68000, 68010, and 68020 Instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
ABCD	D*n*,D*n*	1100 000 100 000 000	6	B		
ABCD	-(A*n*),-(A*n*)	1100 000 100 001 000	6	B		
ADD	#*data*,<*ea1*>	0101 000 000 000 000	24	BWLG	3	ADDQ (Opt)
ADD	#*data*,<*ea3*>	0000 011 000 000 000	25	BWLG		ADDI
ADD	<*ea0*>,D*n*	1101 000 000 000 000	22	BWL		
ADD	D*n*,<*ea2*>	1101 000 100 000 000	23	BWL		
ADD	<*ea0*>,A*n*	1101 000 011 000 000	27	WLG		ADDA
ADDA	#*data*,<*ea1*>	0101 000 000 000 000	24	WLG	3	ADDQ (Opt)
ADDA	<*ea0*>,A*n*	1101 000 011 000 000	27	WL		
ADDI	#*data*,<*ea3*>	0000 011 000 000 000	25	BWL		
ADDQ	#*data*,<*ea1*>	0101 000 000 000 000	24	BWL	3	
ADDX	D*n*,D*n*	1101 000 100 000 000	8	BWL		
ADDX	-(A*n*),-(A*n*)	1101 000 100 001 000	8	BWL		
AND	#*data*,<*ea3*>	0000 001 000 000 000	25	BWLG		ANDI

AND	D*n*, <*ea2*>	1100 000 100 000 000	23	BWL	
AND	<*ea6*>, D*n*	1100 000 000 000 000	22	BWL	
ANDI	#*data*, CCR	0000 001 000 111 100	1	B	8
ANDI	#*data*, SR	0000 001 001 111 100	1	PW	16
ANDI	#*data*, <*ea3*>	0000 001 000 000 000	25	BWL	
ASL	#*data*, D*n*	1110 000 100 000 000	10	BWL	3
ASL	D*n*, D*n*	1110 000 100 100 000	9	BWL	
ASL	<*ea2*>	1110 000 111 000 000	15	W	
ASR	#*data*, D*n*	1110 000 000 000 000	10	BW	3
ASR	D*n*, D*n*	1110 000 000 100 000	9	BWL	
ASR	<*ea2*>	1110 000 011 000 000	15	W	
BCC	*label*	0110 0100 00000000	14	BWL	
BCHG	#*data*, D*n*	0000 100 001 000 000	19	L	
BCHG	#*data*, <*ea2*>	0000 100 001 000 000	19	B	
BCHG	D*n*, D*n*	0000 000 101 000 000	20	L	
BCHG	D*n*, <*ea2*>	0000 000 101 000 000	20	B	
BCLR	#*data*, D*n*	0000 100 010 000 000	19	L	
BCLR	#*data*, <*ea2*>	0000 100 010 000 000	19	B	
BCLR	D*n*, D*n*	0000 000 110 000 000	20	L	
BCLR	D*n*, <*ea2*>	0000 000 110 000 000	20	B	
BCS	*label*	0110 0101 00000000	14	BWL	
BEQ	*label*	0110 0111 00000000	14	BWL	
BFCHG	*special*	1110 101 011 000 000	31	2	
BFCLR	*special*	1110 110 011 000 000	31	2	
BFEXTS	*special*	1110 101 111 000 000	32	2	
BFEXTU	*special*	1110 100 111 000 000	32	2	
BFFFO	*special*	1110 110 111 000 000	32	2	
BFINS	*special*	1110 111 111 000 000	33	2	
BFSET	*special*	1110 111 011 000 000	31	2	
BFTST	*special*	1110 100 011 000 000	31	2	
BGE	*label*	0110 1100 00000000	14	BWL	
BGT	*label*	0110 1110 00000000	14	BWL	
BHI	*label*	0110 0010 00000000	14	BWL	
BHS	*label*	0110 0100 00000000	14	BWLG	BCC
BKPT	#*data*	0100 100 001 001 000	2	2	4

BLE	*label*	0110 1111 00000000	14	BWL	
BLO	*label*	0110 0101 00000000	14	BWLG	BCS
BLS	*label*	0110 0011 00000000	14	BWL	
BLT	*label*	0110 1101 00000000	14	BWL	
BMI	*label*	0110 1011 00000000	14	BWL	
BNE	*label*	0110 0110 00000000	14	BWL	
BNZ	*label*	0110 0110 00000000	14	BWLG	BNE
BPL	*label*	0110 1010 00000000	14	BWL	
BRA	*label*	0110 0000 00000000	14	BWL	
BSET	#*data*,D*n*	0000 100 011 000 000	19	L	
BSET	#*data*,<*ea2*>	0000 100 011 000 000	19	B	
BSET	D*n*,D*n*	0000 000 111 000 000	20	L	
BSET	D*n*,<*ea2*>	0000 000 111 000 000	20	B	
BSR	*label*	0110 0001 00000000	14	BWL	
BT	*label*	0110 0000 00000000	14	BWLG	BRA
BTST	#*data*,D*n*	0000 100 000 000 000	19	L	
BTST	#*data*,<*ea5*>	0000 100 000 000 000	19	B	
BTST	D*n*,D*n*	0000 000 100 000 000	20	L	
BTST	D*n*,<*ea12*>	0000 000 100 000 000	20	B	
BVC	*label*	0110 1000 00000000	14	BWL	
BVS	*label*	0110 1001 00000000	14	BWL	
BZ	*label*	0110 0111 00000000	14	BWL	BEQ
CALLM	#*data*,<*ea7*>	0000 011 011 000 000	47	2	
CAS	*special*	0000 100 011 000 000	34	2BWL	
CAS2	*special*	0000 100 011 111 100	35	2BWL	
CHK	<*ea6*>,D*n*	0100 000 110 000 000	21	W	
CHK	<*ea6*>,D*n*	0100 000 100 000 000	21	2L	
CHK2	<*ea7*>,D*n*	0000 000 011 000 000	36	2BWL	
CHK2	<*ea7*>,A*n*	0000 000 011 000 000	36	2BWL	
CLR	D*n*	0111 000 0 00000000	11	LG	MOVEQ (Opt)
CLR	<*ea3*>	0100 001 000 000 000	17	BWL	
CMP	#*data*,<*ea3*>	0000 110 000 000 000	25	BWLG	CMPI
CMP	<*ea0*>,D*n*	1011 000 000 000 000	22	BWL	
CMP	<*ea0*>,A*n*	1011 000 011 000 000	27	WLG	CMPA

CMP	(A*n*)+, (A*n*)+	1011 000 100 001 000	6	WG	CMPM
CMP	(A*n*)+, (A*n*)+	1011 000 101 001 000	6	BG	CMPM
CMP	(A*n*)+, (A*n*)+	1011 000 110 001 000	6	LG	CMPM
CMPA	<*ea0*>, A*n*	1011 000 011 000 000	27	WL	
CMPI	#*data*, <*ea3*>	0000 110 000 000 000	25	BWL	
CMPM	(A*n*)+, (A*n*)+	1011 000 100 001 000	6	W	
CMPM	(A*n*)+, (A*n*)+	1011 000 101 001 000	6	B	
CMPM	(A*n*)+, (A*n*)+	1011 000 110 001 000	6	L	
CMP2	<*ea7*>, D*n*	0000 000 011 000 000	37	2BWL	
CMP2	<*ea7*>, A*n*	0000 000 011 000 000	37	2BWL	
DBCC	D*n*, *label*	0101 0100 11 001 000	13	W	
DBCS	D*n*, *label*	0101 0101 11 001 000	13	W	
DBEQ	D*n*, *label*	0101 0111 11 001 000	13	W	
DBF	D*n*, *label*	0101 0001 11 001 000	13	W	
DBGE	D*n*, *label*	0101 1100 11 001 000	13	W	
DBGT	D*n*, *label*	0101 1110 11 001 000	13	W	
DBHI	D*n*, *label*	0101 0010 11 001 000	13	W	
DBHS	D*n*, *label*	0101 0100 11 001 000	13	WG	DBCC
DBLE	D*n*, *label*	0101 1111 11 001 000	13	W	
DBLO	D*n*, *label*	0101 0101 11 001 000	13	WG	DBCS
DBLS	D*n*, *label*	0101 0011 11 001 000	13	W	
DBLT	D*n*, *label*	0101 1101 11 001 000	13	W	
DBMI	D*n*, *label*	0101 1011 11 001 000	13	W	
DBNE	D*n*, *label*	0101 0110 11 001 000	13	W	
DBNZ	D*n*, *label*	0101 0110 11 001 000	13	WG	DBNE
DBPL	D*n*, *label*	0101 1010 11 001 000	13	W	
DBRA	D*n*, *label*	0101 0001 11 001 000	13	WG	DBF
DBT	D*n*, *label*	0101 0000 11 001 000	13	W	
DBVC	D*n*, *label*	0101 1000 11 001 000	13	W	
DBVS	D*n*, *label*	0101 1001 11 001 000	13	W	
DBZ	D*n*, *label*	0101 0111 11 001 000	13	W	DBEQ
DIVS	*special*	0100 110 001 000 000	38	2L	
DIVS	<*ea6*>, D*n*	1000 000 111 000 000	21	W	

DIVU	*special*	0100 110 001 000 000	39	2L	
DIVU	<*ea6*>,D*n*	1000 000 011 000 000	21	W	
EOR	#*data*,<*ea3*>	0000 101 000 000 000	25	BWLG	EORI
EOR	D*n*,<*ea3*>	1011 000 100 000 000	23	BWL	
EORI	#*data*,<*ea3*>	0000 101 000 000 000	25	BWL	
EORI	#*data*,CCR	0000 101 000 111 100	1	B 8	
EORI	#*data*,SR	0000 101 001 111 100	1	PW 16	
EXG	A*n*,D*n*	1100 000 110 001 000	6	L	
EXG	D*n*,D*n*	1100 000 101 000 000	7	L	
EXG	A*n*,A*n*	1100 000 101 001 000	7	L	
EXG	D*n*,A*n*	1100 000 110 001 000	7	L	
EXT	D*n*	0100 100 010 000 000	2	W	
EXT	D*n*	0100 100 011 000 000	2	L	
EXTB	D*n*	0100 100 010 000 000	2	WG	EXT.W
EXTB	D*n*	0100 100 111 000 000	2	2L	
EXTW	D*n*	0100 100 011 000 000	2	LG	EXT.L
JMP	<*ea7*>	0100 111 011 000 000	15	W	
JSR	<*ea7*>	0100 111 010 000 000	15	W	
ILLEGAL		0100 101 011 111 100	0		
LEA	<*ea7*>,A*n*	0100 000 111 000 000	46	L	
LINK	A*n*,#*data*	0100 111 001 010 000	4	W -16	
LINK	A*n*,#*data*	0100 100 000 001 000	4	L	
LSL	#*data*,D*n*	1110 000 100 001 000	10	BWL 3	
LSL	D*n*,D*n*	1110 000 100 101 000	9	BWL	
LSL	<*ea2*>	1110 001 111 000 000	15	W	
LSR	#*data*,D*n*	1110 000 000 001 000	10	BWL 3	
LSR	D*n*,D*n*	1110 000 000 101 000	9	BWL	
LSR	<*ea2*>	1110 001 011 000 000	15	W	
MOVE	#*data*,D*n*	0111 000 0 00000000	11	LG -8	MOVEQ (Opt)
MOVE	#*data*,<*ea3*>	0100 001 000 000 000	18	BWLC 0	CLR (Opt)
MOVE	#*data*,A*n*	1001 000 111 000 000	28	WLG 0	SUBA.L (Opt)
MOVE	<*ea0*>,<*ea3*>	0000 000 000 000 000	26	BWL	
MOVE	<*ea0*>,A*n*	0000 000 000 000 000	26	WLG	MOVEA
MOVE	A*n*,USP	0100 111 001 100 000	2	PL	
MOVE	USP,A*n*	0100 111 001 101 000	3	PL	
MOVE	<*ea6*>,CCR	0100 011 011 000 000	15	W	
MOVE	CCR,<*ea3*>	0100 001 011 000 000	16	1 W	
MOVE	<*ea6*>,SR	0100 010 011 000 000	15	PW	

MOVE	SR, *<ea3>*	0100 000 011 000 000	16	PW	
MOVEA	#*data*, A*n*	1001 000 111 000 000	28	WLG	0 SUBA.L (Opt)
MOVEA	*<ea0>*, A*n*	0000 000 000 000 000	26	WL	
MOVEC	Rc, D*n*	0100 111 001 111 010	42	1PL	
MOVEC	Rc, A*n*	0100 111 001 111 010	42	1PL	
MOVEC	D*n*, Rc	0100 111 001 111 011	42	1PL	
MOVEC	A*n*, Rc	0100 111 001 111 011	42	1PL	
MOVEM	*<ea8>, rlist*	0100 110 010 000 000	29	WL	
MOVEM	*rlist, <ea4>*	0100 100 010 000 000	30	WL	
MOVEM	*<ea8>*, A*n*	0000 000 000 000 000	26	WLG	MOVEA
MOVEP	d(A*n*), D*n*	0000 000 100 001 000	5	W	
MOVEP	D*n*, d(A*n*)	0000 000 110 001 000	5	W	
MOVEP	d(A*n*), D*n*	0000 000 101 001 000	5	L	
MOVEP	D*n*, d(A*n*)	0000 000 111 001 000	5	L	
MOVEQ	#*data*, D*n*	0111 000 0 00000000	11	L	-8
MOVEQ	#*data*, A*n*	1001 000 111 000 000	28	LG	0 SUBA.L (Opt)
MOVES	D*n*, *<ea2>*	0000 111 000 000 000	43	1PBWL	
MOVES	A*n*, *<ea2>*	0000 111 000 000 000	43	1PBWL	
MOVES	*<ea2>*, D*n*	0000 111 000 000 000	43	1PBWL	
MOVES	*<ea2>*, A*n*	0000 111 000 000 000	43	1PBWL	
MULS	*special*	0100 110 000 000 000	38	2L	
MULS	*<ea6>*, D*n*	1100 000 111 000 000	21	W	
MULU	*special*	0100 110 000 000 000	39	2L	
MULU	*<ea6>*, D*n*	1100 000 011 000 000	21	W	
NBCD	*<ea3>*	0100 100 000 000 000	15	B	
NEG	*<ea3>*	0100 010 000 000 000	17	BWL	
NEGX	*<ea3>*	0100 000 000 000 000	17	BWL	
NOP		0100 111 001 110 001	0		
NOT	*<ea3>*	0100 011 000 000 000	17	BWL	
OR	#*data*, *<ea3>*	0000 000 000 000 000	25	BWLG	ORI
OR	*<ea6>*, D*n*	1000 000 000 000 000	22	BWL	
OR	D*n*, *<ea2>*	1000 000 100 000 000	23	BWL	
ORI	#*data*, *<ea3>*	0000 000 000 000 000	25	BWL	
ORI	#*data*, CCR	0000 000 000 111 100	1	B	8
ORI	#*data*, SR	0000 000 001 111 100	1	PW	16
PACK	*special*	1000 000 101 000 000	44	2	16
PEA	*<ea7>*	0100 100 001 000 000	45	L	

RESET		0100 111 001 110 000	0	P	
ROL	#*data*,D*n*	1110 000 100 011 000	10	BWL	3
ROL	D*n*,D*n*	1110 000 100 111 000	9	BWL	
ROL	<*ea2*>	1110 011 011 000 000	15	W	
ROR	#*data*,D*n*	1110 000 000 011 000	10	BWL	3
ROR	D*n*,D*n*	1110 000 000 111 000	9	BWL	
ROR	<*ea2*>	1110 011 111 000 000	15	W	
ROXL	#*data*,D*n*	1110 000 100 010 000	10	BWL	3
ROXL	D*n*,D*n*	1110 000 100 110 000	9	BWL	
ROXL	<*ea2*>	1110 010 111 000 000	15	W	
ROXR	#*data*,D*n*	1110 000 000 010 000	10	BWL	3
ROXR	D*n*,D*n*	1110 000 000 110 000	9	BWL	
ROXR	<*ea2*>	1110 010 011 000 000	15	W	
RTD	#*data*	0100 111 001 110 100	1	1W	-16
RTE		0100 111 001 110 011	0	P	
RTM	D*n*	0000 011 011 000 000	48	2	
RTM	A*n*	0000 011 011 001 000	48	2	
RTR		0100 111 001 110 111	0		
RTS		0100 111 001 110 101	0		
SBCD	D*n*,D*n*	1000 000 100 000 000	6	B	
SBCD	-(A*n*),-(A*n*)	1000 000 100 001 000	6	B	
SCC	<*ea3*>	0101 0100 11 000 000	15	B	
SCS	<*ea3*>	0101 0101 11 000 000	15	B	
SEQ	<*ea3*>	0101 0111 11 000 000	15	B	
SF	<*ea3*>	0101 0001 11 000 000	15	B	
SGE	<*ea3*>	0101 1100 11 000 000	15	B	
SGT	<*ea3*>	0101 1110 11 000 000	15	B	
SHI	<*ea3*>	0101 0010 11 000 000	15	B	
SHS	<*ea3*>	0101 0100 11 000 000	15	BG	SCC
SLE	<*ea3*>	0101 1111 11 000 000	15	B	
SLO	<*ea3*>	0101 0101 11 000 000	15	BG	SCS
SLS	<*ea3*>	0101 0011 11 000 000	15	B	
SLT	<*ea3*>	0101 1101 11 000 000	15	B	
SMI	<*ea3*>	0101 1011 11 000 000	15	B	

SNE	*<ea3>*	0101 0110 11 000 000	15	B		
SNZ	*<ea3>*	0101 0110 11 000 000	15	BG		SNE
SPL	*<ea3>*	0101 1010 11 000 000	15	B		
ST	*<ea3>*	0101 0000 11 000 000	15	B		
STOP	*#data*	0100 111 001 110 010	1	W	16	
SUB	*#data, <ea1>*	0101 000 100 000 000	24	BWLG	3	SUBQ (Opt)
SUB	*#data, <ea3>*	0000 010 000 000 000	25	BWLG		SUBI
SUB	D*n*, *<ea2>*	1001 000 100 000 000	23	BWL		
SUB	*<ea0>*, D*n*	1001 000 000 000 000	22	BWL		
SUB	*<ea0>*, A*n*	1001 000 011 000 000	27	WLG		SUBA
SUBA	*#data, <ea1>*	0101 000 100 000 000	24	WLG	3	SUBQ (Opt)
SUBA	*<ea0>*, A*n*	1001 000 011 000 000	27	WL		
SUBI	*#data, <ea3>*	0000 010 000 000 000	25	BWL		
SUBQ	*#data, <ea1>*	0101 000 100 000 000	24	BWL	3	
SUBX	D*n*,D*n*	1001 000 100 000 000	8	BWL		
SUBX	-(A*n*),-(A*n*)	1001 000 100 001 000	8	BWL		
SVC	*<ea3>*	0101 1000 11 000 000	15	B		
SVS	*<ea3>*	0101 1001 11 000 000	15	B		
SWAP	D*n*	0100 100 001 000 000	2	W		
TAS	*<ea3>*	0100 101 011 000 000	15	B		
TCC		0101 0100 11111 100	0	2		
TCS		0101 0101 11111 100	0	2		
TDIVS	*special*	0100 110 001 000 000	40	2L		
TDIVU	*special*	0100 110 001 000 000	41	2L		
TEQ		0101 0111 11111 100	0	2		
TF		0101 0001 11111 100	0	2		
TGE		0101 1100 11111 100	0	2		
TGT		0101 1110 11111 100	0	2		
THI		0101 0010 11111 100	0	2		
THS		0101 0100 11111 100	0	2G		TCC
TLE		0101 1111 11111 100	0	2		
TLO		0101 0101 11111 100	0	2G		TCS

TLS		0101 0011 11111 100	0	2			
TLT		0101 1101 11111 100	0	2			
TMI		0101 1011 11111 100	0	2			
TNE		0101 0110 11111 100	0	2			
TNZ		0101 0110 11111 100	0	2G		TNE	
TPCC	#*data*	0101 0100 11111 010	1	2W	-16		
TPCC	#*data*	0101 0100 11111 011	1	2L			
TPCS	#*data*	0101 0101 11111 010	1	2W	-16		
TPCS	#*data*	0101 0101 11111 011	1	2L			
TPEQ	#*data*	0101 0111 11111 010	1	2W	-16		
TPEQ	#*data*	0101 0111 11111 011	1	2L			
TPF	#*data*	0101 0001 11111 010	1	2W	-16		
TPF	#*data*	0101 0001 11111 011	1	2L			
TPGE	#*data*	0101 1100 11111 010	1	2W	-16		
TPGE	#*data*	0101 1100 11111 011	1	2L			
TPGT	#*data*	0101 1110 11111 010	1	2W	-16		
TPGT	#*data*	0101 1110 11111 011	1	2L			
TPHI	#*data*	0101 0010 11111 010	1	2W	-16		
TPHI	#*data*	0101 0010 11111 011	1	2L			
TPHS	#*data*	0101 0100 11111 010	1	2WG	-16	TPCC	
TPHS	#*data*	0101 0100 11111 011	1	2LG		TPCC	
TPL		0101 1010 11111 100	0	2			
TPLE	#*data*	0101 1111 11111 010	1	2W	-16		
TPLE	#*data*	0101 1111 11111 011	1	2L			
TPLO	#*data*	0101 0101 11111 010	1	2WG	-16	TPCS	
TPLO	#*data*	0101 0101 11111 011	1	2LG		TPCS	
TPLS	#*data*	0101 0011 11111 010	1	2W	-16		
TPLS	#*data*	0101 0011 11111 011	1	2L			
TPLT	#*data*	0101 1101 11111 010	1	2W	-16		
TPLT	#*data*	0101 1101 11111 011	1	2L			
TPMI	#*data*	0101 1011 11111 010	1	2W	-16		
TPMI	#*data*	0101 1011 11111 011	1	2L			
TPNE	#*data*	0101 0110 11111 010	1	2W	-16		
TPNE	#*data*	0101 0110 11111 011	1	2L			
TPNZ	#*data*	0101 0110 11111 010	1	2WG	-16	TPNE	

TPNZ	#*data*	0101 0110 11111 011	1	2LG		TPNE
TPPL	#*data*	0101 1010 11111 010	1	2W	-16	
TPPL	#*data*	0101 1010 11111 011	1	2L		
TPT	#*data*	0101 0000 11111 010	1	2W	-16	
TPT	#*data*	0101 0000 11111 011	1	2L		
TPVC	#*data*	0101 1000 11111 010	1	2W	-16	
TPVC	#*data*	0101 1000 11111 011	1	2L		
TPVS	#*data*	0101 1001 11111 010	1	2W	-16	
TPVS	#*data*	0101 1001 11111 011	1	2L		
TPZ	#*data*	0101 0111 11111 010	1	2W	-16	TPEQ
TPZ	#*data*	0101 0111 11111 011	1	2L		TPEQ
TRAP	#*data*	0100 111 001 00 0000	12		4	
TRAPV		0100 111 001 110 110	0			
TST	<*ea3*>	0100 101 000 000 000	17	BWL		
TT		0101 0000 11111 100	0	2		
TVC		0101 1000 11111 100	0	2		
TVS		0101 1001 11111 100	0	2		
TZ		0101 0111 11111 100	0	2		TEQ
UNLK	A*n*	0100 111 001 011 000	2			
UNPK	*special*	1000 000 110 000 000	44	2	16	

Table G-8
MC68881 instructions

Opcode	Operands	Opcode Word	cp-Type	Group	Flags	Equivalent
FABS	FP*n*	000 000 000 0011000	Gen1	1	X	
FABS	FP*n*,FP*n*	000 000 000 0011000	Gen1	2	X	
FABS	<*ea6*>,FP*n*	010 000 000 0011000	Gen2	3	BWLSDXK	
FACOS	FP*n*	000 000 000 0011100	Gen1	1	X	
FACOS	FP*n*,FP*n*	000 000 000 0011100	Gen1	2	X	
FACOS	<*ea6*>,FP*n*	010 000 000 0011100	Gen2	3	BWLSDXK	
FADD	FP*n*	000 000 000 0100010	Gen1	1	XG	FADD FP*n*,FP*n*
FADD	FP*n*,FP*n*	000 000 000 0100010	Gen1	2	X	
FADD	<*ea6*>,FP*n*	010 000 000 0100010	Gen2	3	BWLSDXK	
FASIN	FP*n*	000 000 000 0001100	Gen1	1	X	
FASIN	FP*n*,FP*n*	000 000 000 0001100	Gen1	2	X	
FASIN	<*ea6*>,FP*n*	010 000 000 0001100	Gen2	3	BWLSDXK	
FATAN	FP*n*	000 000 000 0001010	Gen1	1	X	
FATAN	FP*n*,FP*n*	000 000 000 0001010	Gen1	2	X	
FATAN	<*ea6*>,FP*n*	010 000 000 0001010	Gen2	3	BWLSDXK	
FATANH	FP*n*	000 000 000 0001101	Gen1	1	X	
FATANH	FP*n*,FP*n*	000 000 000 0001101	Gen1	2	X	
FATANH	<*ea6*>,FP*n*	010 000 000 0001101	Gen2	3	BWLSDXK	
FBEQ	*label*	1111 000 01 0 000001	Bcc	4	WL	
FBF	*label*	1111 000 01 0 000000	Bcc	4	WL	
FBGE	*label*	1111 000 01 0 010011	Bcc	4	WL	
FBGL	*label*	1111 000 01 0 010110	Bcc	4	WL	
FBGLE	*label*	1111 000 01 0 010111	Bcc	4	WL	
FBGT	*label*	1111 000 01 0 010010	Bcc	4	WL	
FBLE	*label*	1111 000 01 0 010101	Bcc	4	WL	
FBLT	*label*	1111 000 01 0 010100	Bcc	4	WL	
FBNE	*label*	1111 000 01 0 001110	Bcc	4	WL	
FBNGE	*label*	1111 000 01 0 011100	Bcc	4	WL	
FBNGL	*label*	1111 000 01 0 011001	Bcc	4	WL	
FBNGLE	*label*	1111 000 01 0 011000	Bcc	4	WL	
FBNGT	*label*	1111 000 01 0 011101	Bcc	4	WL	
FBNLE	*label*	1111 000 01 0 011010	Bcc	4	WL	
FBNLT	*label*	1111 000 01 0 011011	Bcc	4	WL	
FBOGE	*label*	1111 000 01 0 000011	Bcc	4	WL	
FBOGL	*label*	1111 000 01 0 000110	Bcc	4	WL	
FBOGT	*label*	1111 000 01 0 000010	Bcc	4	WL	
FBOLE	*label*	1111 000 01 0 000101	Bcc	4	WL	
FBOLT	*label*	1111 000 01 0 000111	Bcc	4	WL	
FBRA	*label*	1111 000 01 0 001111	Bcc	4	WLGFBT	
FBSEQ	*label*	1111 000 01 0 010001	Bcc	4	WL	

FBSF	*label*	1111 000 01 0 010000	Bcc	4	WL
FBSNE	*label*	1111 000 01 0 011110	Bcc	4	WL
FBST	*label*	1111 000 01 0 011111	Bcc	4	WL
FBT	*label*	1111 000 01 0 001111	Bcc	4	WL
FBUEQ	*label*	1111 000 01 0 001001	Bcc	4	WL
FBUGE	*label*	1111 000 01 0 001011	Bcc	4	WL
FBUGT	*label*	1111 000 01 0 001010	Bcc	4	WL
FBULE	*label*	1111 000 01 0 001101	Bcc	4	WL
FBULT	*label*	1111 000 01 0 001100	Bcc	4	WL
FBUN	*label*	1111 000 01 0 001000	Bcc	4	WL
FCMP	FP*n*,FP*n*	000 000 000 0111000	Gen1	2	X
FCMP	<*ea6*>,FP*n*	010 000 000 0111000	Gen2	3	BWLSDXK
FCOS	FP*n*	000 000 000 0011101	Gen1	1	X
FCOS	FP*n*,FP*n*	000 000 000 0011101	Gen1	2	X
FCOS	<*ea6*>,FP*n*	010 000 000 0011101	Gen2	3	BWLSDXK
FCOSH	FP*n*	000 000 000 0011001	Gen1	1	X
FCOSH	FP*n*,FP*n*	000 000 000 0011001	Gen1	2	X
FCOSH	<*ea6*>,FP*n*	010 000 000 0011001	Gen2	3	BWLSDXK
FDBEQ	D*n*,*label*	0000000000 000001	DBcc	5	W
FDBF	D*n*,*label*	0000000000 000000	DBcc	5	W
FDBGE	D*n*,*label*	0000000000 010011	DBcc	5	W
FDBGL	D*n*,*label*	0000000000 010110	DBcc	5	W
FDBGLE	D*n*,*label*	0000000000 010111	DBcc	5	W
FDBGT	D*n*,*label*	0000000000 010010	DBcc	5	W
FDBLE	D*n*,*label*	0000000000 010101	DBcc	5	W
FDBLT	D*n*,*label*	0000000000 010100	DBcc	5	W
FDBNE	D*n*,*label*	0000000000 001110	DBcc	5	W
FDBNGE	D*n*,*label*	0000000000 011100	DBcc	5	W
FDBNGL	D*n*,*label*	0000000000 011001	DBcc	5	W
FDBNGLE	D*n*,*label*	0000000000 011000	DBcc	5	W
FDBNGT	D*n*,*label*	0000000000 011101	DBcc	5	W
FDBNLE	D*n*,*label*	0000000000 011010	DBcc	5	W
FDBNLT	D*n*,*label*	0000000000 011011	DBcc	5	W
FDBOGE	D*n*,*label*	0000000000 000011	DBcc	5	W
FDBOGL	D*n*,*label*	0000000000 000110	DBcc	5	W
FDBOGT	D*n*,*label*	0000000000 000010	DBcc	5	W
FDBOLE	D*n*,*label*	0000000000 000101	DBcc	5	W
FDBOLT	D*n*,*label*	0000000000 000100	DBcc	5	W
FDBOR	D*n*,*label*	0000000000 000111	DBcc	5	W
FDBRA	D*n*,*label*	0000000000 001111	DBcc	5	WGFDBT
FDBSEQ	D*n*,*label*	0000000000 010001	DBcc	5	W
FDBSF	D*n*,*label*	0000000000 010000	DBcc	5	W
FDBSNE	D*n*,*label*	0000000000 011110	DBcc	5	W
FDBST	D*n*,*label*	0000000000 011111	DBcc	5	W
FDBT	D*n*,*label*	0000000000 001111	DBcc	5	W

FDBUEQ	*Dn,label*	0000000000 001001	DBcc	5	W	
FDBUGE	*Dn,label*	0000000000 001011	DBcc	5	W	
FDBUGT	*Dn,label*	0000000000 001010	DBcc	5	W	
FDBULE	*Dn,label*	0000000000 001101	DBcc	5	W	
FDBULT	*Dn,label*	0000000000 001100	DBcc	5	W	
FDBUN	*Dn,label*	0000000000 001000	DBcc	5	W	
FDIV	*FPn,FPn*	000 000 000 0100000	Gen1	2	X	
FDIV	*<ea6>,FPn*	010 000 000 0100000	Gen2	3	BWLSDXK	
FETOX	*FPn*	000 000 000 0010000	Gen1	1	X	
FETOX	*FPn,FPn*	000 000 000 0010000	Gen1	2	X	
FETOX	*<ea6>,FPn*	010 000 000 0010000	Gen2	3	BWLSDXK	
FETOXM1	*FPn*	000 000 000 0001000	Gen1	1	X	
FETOXM1	*FPn,FPn*	000 000 000 0001000	Gen1	2	X	
FETOXM1	*<ea6>,FPn*	010 000 000 0001000	Gen2	3	BWLSDXK	
FGETEXP	*FPn*	000 000 000 0011110	Gen1	1	X	
FGETEXP	*FPn,FPn*	000 000 000 0011110	Gen1	2	X	
FGETEXP	*<ea6>,FPn*	010 000 000 0011110	Gen2	3	BWLSDXK	
FGETMAN	*FPn*	000 000 000 0011111	Gen1	1	X	
FGETMAN	*FPn,FPn*	000 000 000 0011111	Gen1	2	X	
FGETMAN	*<ea6>,FPn*	010 000 000 0011111	Gen2	3	BWLSDXK	
FINT	*FPn*	000 000 000 0000001	Gen1	1	X	
FINT	*FPn,FPn*	000 000 000 0000001	Gen1	2	X	
FINT	*<ea6>,FPn*	010 000 000 0000001	Gen2	3	BWLSDXK	
FINTRZ	*FPn*	000 000 000 0000011	Gen1	1	X	
FINTRZ	*FPn,FPn*	000 000 000 0000011	Gen1	2	X	
FINTRZ	*<ea6>,FPn*	010 000 000 0000011	Gen2	3	BWLSDXK	
FLOG10	*FPn*	000 000 000 0010101	Gen1	1	X	
FLOG10	*FPn,FPn*	000 000 000 0010101	Gen1	2	X	
FLOG10	*<ea6>,FPn*	010 000 000 0010101	Gen2	3	BWLSDXK	
FLOG2	*FPn*	000 000 000 0010110	Gen1	1	X	
FLOG2	*FPn,FPn*	000 000 000 0010110	Gen1	2	X	
FLOG2	*<ea6>,FPn*	010 000 000 0010110	Gen2	3	BWLSDXK	
FLOGN	*FPn*	000 000 000 0010100	Gen1	1	X	
FLOGN	*FPn,FPn*	000 000 000 0010100	Gen1	2	X	
FLOGN	*<ea6>,FPn*	010 000 000 0010100	Gen2	3	BWLSDXK	
FLOGNP1	*FPn*	000 000 000 0000110	Gen1	1	X	
FLOGNP1	*FPn,FPn*	000 000 000 0000110	Gen1	2	X	
FLOGNP1	*<ea6>,FPn*	010 000 000 0000110	Gen2	3	BWLSDXK	
FMOD	*FPn,FPn*	000 000 000 0100001	Gen1	2	X	
FMOD	*<ea6>,FPn*	010 000 000 0100001	Gen2	3	BWLSDXK	
FMOVE	*special*	011 000 000 0000000		21	P	{k-factor}
FMOVE	*FPn,FPn*	000 000 000 0000000	Gen1	2	X	
FMOVE	*FPn,<ea3>*	011 000 000 0000000	Gen3	6	BWLSDX	
FMOVE	*<ea6>,FPn*	010 000 000 0000000	Gen2	3	BWLSDXK	
FMOVE	*FPCR,<ea3>*	101 100 0000000000	Gen3	7	L	

FMOVE	FPSR,<ea3>	101 010 0000000000	Gen3	7	L	
FMOVE	FPIAR,<ea3>	101 001 0000000000	Gen3	7	L	
FMOVE	FPIAR,An	101 001 0000000000	Gen3	7	L	
FMOVE	<ea6>,FPCR	100 100 0000000000	Gen2	8	L	
FMOVE	<ea6>,FPSR	100 010 0000000000	Gen2	8	L	
FMOVE	<ea6>,FPIAR	100 001 0000000000	Gen2	8	L	
FMOVE	An,FPIAR	100 001 0000000000	Gen2	8	L	
FMOVECR	#data,FPn	010111 000 0000000	Gen1	9	X	
FMOVEM	<ea8>,frlist	110 10 000 00000000	Gen2	10	X	
FMOVEM	frlist,<ea4>	111 00 000 00000000	Gen3	11	X	
FMOVEM	<ea8>,Dn	110 11 000 00000000	Gen2	12	X	
FMOVEM	Dn,<ea4>	111 01 000 00000000	Gen3	13	X	
FMOVEM	<ea0>,fcrlist	11 0 000 0000000000	Gen2	14	L	
FMOVEM	fcrlist,<ea1>	11 1 000 0000000000	Gen3	15	L	
FMUL	FPn	000 000 000 0100011	Gen1	1	XG	FMUL FPn,FPn
FMUL	FPn,FPn	000 000 000 0100011	Gen1	2	X	
FMUL	<ea6>,FPn	010 000 000 0100011	Gen2	3	BWLSDXK	
FNEG	FPn	000 000 000 0011010	Gen1	1	X	
FNEG	FPn,FPn	000 000 000 0011010	Gen1	2	X	
FNEG	<ea6>,FPn	010 000 000 0011010	Gen2	3	BWLSDXK	
FNOP		0000000000000000		0		
FREM	FPn,FPn	000 000 000 0100101	Gen1	2	X	
FREM	<ea6>,FPn	010 000 000 0100101	Gen2	3	BWLSDXK	
FRESTORE	<ea8>	1111 000 101 000 000	Rest	16	P	
FSAVE	<ea4>	1111 000 100 000 000	Save	16	P	
FSCALE	FPn,FPn	000 000 000 0100110	Gen1	2	X	
FSCALE	<ea6>,FPn	010 000 000 0100110	Gen2	3	BWLSDXK	
FSEQ	<ea3>	0000000000 000001	Scc	17	B	
FSF	<ea3>	0000000000 000000	Scc	17	B	
FSGE	<ea3>	0000000000 010011	Scc	17	B	
FSGL	<ea3>	0000000000 010110	Scc	17	B	
FSGLE	<ea3>	0000000000 010111	Scc	17	B	
FSGT	<ea3>	0000000000 010010	Scc	17	B	
FSLE	<ea3>	0000000000 010101	Scc	17	B	
FSLT	<ea3>	0000000000 010100	Scc	17	B	
FSNE	<ea3>	0000000000 001110	Scc	17	B	
FSNGE	<ea3>	0000000000 011100	Scc	17	B	
FSNGL	<ea3>	0000000000 011001	Scc	17	B	
FSNGLE	<ea3>	0000000000 011000	Scc	17	B	
FSNGT	<ea3>	0000000000 011101	Scc	17	B	
FSNLE	<ea3>	0000000000 011010	Scc	17	B	
FSNLT	<ea3>	0000000000 011011	Scc	17	B	
FSOGE	<ea3>	0000000000 000011	Scc	17	B	
FSOGL	<ea3>	0000000000 000110	Scc	17	B	
FSOGT	<ea3>	0000000000 000010	Scc	17	B	

FSOLE	<ea3>	0000000000 000101	Scc	17	B
FSOLT	<ea3>	0000000000 000100	Scc	17	B
FSOR	<ea3>	0000000000 000111	Scc	17	B
FSSEQ	<ea3>	0000000000 010001	Scc	17	B
FSSF	<ea3>	0000000000 010000	Scc	17	B
FSSNE	<ea3>	0000000000 011110	Scc	17	B
FSST	<ea3>	0000000000 011111	Scc	17	B
FST	<ea3>	0000000000 001111	Scc	17	B
FSUEQ	<ea3>	0000000000 001001	Scc	17	B
FSUGE	<ea3>	0000000000 001011	Scc	17	B
FSUGT	<ea3>	0000000000 001010	Scc	17	B
FSULE	<ea3>	0000000000 001101	Scc	17	B
FSULT	<ea3>	0000000000 001100	Scc	17	B
FSUN	<ea3>	0000000000 001000	Scc	17	B
FSGLDIV	FPn,FPn	000 000 000 0100100	Gen1	2	X
FSGLDIV	<ea6>,FPn	010 000 000 0100100	Gen2	3	BWLSDXK
FSGLMUL	FPn,FPn	000 000 000 0100111	Gen1	2	X
FSGLMUL	<ea6>,FPn	010 000 000 0100111	Gen2	3	BWLSDXK
FSIN	FPn	000 000 000 0001110	Gen1	1	X
FSIN	FPn,FPn	000 000 000 0001110	Gen1	2	X
FSIN	<ea6>,FPn	010 000 000 0001110	Gen2	3	BWLSDXK
FSINCOS	special	000 000 000 0110 000		22	BWLSDXK
FSINH	FPn	000 000 000 0000010	Gen1	1	X
FSINH	FPn,FPn	000 000 000 0000010	Gen1	2	X
FSINH	<ea6>,FPn	010 000 000 0000010	Gen2	3	BWLSDXK
FSQRT	FPn	000 000 000 0000100	Gen1	1	X
FSQRT	FPn,FPn	000 000 000 0000100	Gen1	2	X
FSQRT	<ea6>,FPn	010 000 000 0000100	Gen2	3	BWLSDXK
FSUB	FPn,FPn	000 000 000 0101000	Gen1	2	X
FSUB	<ea6>,FPn	010 000 000 0101000	Gen2	3	BWLSDXK
FTAN	FPn	000 000 000 0001111	Gen1	1	X
FTAN	FPn,FPn	000 000 000 0001111	Gen1	2	X
FTAN	<ea6>,FPn	010 000 000 0001111	Gen2	3	BWLSDXK
FTANH	FPn	000 000 000 0001001	Gen1	1	X
FTANH	FPn,FPn	000 000 000 0001001	Gen1	2	X
FTANH	<ea6>,FPn	010 000 000 0001001	Gen2	3	BWLSDXK
FTENTOX	FPn	000 000 000 0010010	Gen1	1	X
FTENTOX	FPn,FPn	000 000 000 0010010	Gen1	2	X
FTENTOX	<ea6>,FPn	010 000 000 0010010	Gen2	3	BWLSDXK
FTEST	FPn	000 000 000 0111010	Gen1	20	X
FTEST	<ea6>	010 000 000 0111010	Gen2	20	BWLSDXK
FTEQ		0000000000 000001	Tcc1	18	
FTF		0000000000 000000	Tcc1	18	
FTGE		0000000000 010011	Tcc1	18	
FTGL		0000000000 010110	Tcc1	18	

FTGLE		0000000000 010111	Tcc1	18	
FTGT		0000000000 010010	Tcc1	18	
FTLE		0000000000 010101	Tcc1	18	
FTLT		0000000000 010100	Tcc1	18	
FTNE		0000000000 001110	Tcc1	18	
FTNGE		0000000000 011100	Tcc1	18	
FTNGL		0000000000 011001	Tcc1	18	
FTNGLE		0000000000 011000	Tcc1	18	
FTNGT		0000000000 011101	Tcc1	18	
FTNLE		0000000000 011010	Tcc1	18	
FTNLT		0000000000 011011	Tcc1	18	
FTOGE		0000000000 000011	Tcc1	18	
FTOGL		0000000000 000110	Tcc1	18	
FTOGT		0000000000 000010	Tcc1	18	
FTOLE		0000000000 000101	Tcc1	18	
FTOLT		0000000000 000100	Tcc1	18	
FTOR		0000000000 000111	Tcc1	18	
FTSEQ		0000000000 010001	Tcc1	18	
FTSF		0000000000 010000	Tcc1	18	
FTSNE		0000000000 011110	Tcc1	18	
FTST		0000000000 011111	Tcc1	18	
FTT		0000000000 001111	Tcc1	18	
FTUEQ		0000000000 001001	Tcc1	18	
FTUGE		0000000000 001011	Tcc1	18	
FTUGT		0000000000 001010	Tcc1	18	
FTULE		0000000000 001101	Tcc1	18	
FTULT		0000000000 001100	Tcc1	18	
FTUN		0000000000 001000	Tcc1	18	
FTPEQ	*#data*	0000000000 000001	Tcc2	19	WL
FTPF	*#data*	0000000000 000000	Tcc2	19	WL
FTPGE	*#data*	0000000000 010011	Tcc2	19	WL
FTPGL	*#data*	0000000000 010110	Tcc2	19	WL
FTPGLE	*#data*	0000000000 010111	Tcc2	19	WL
FTPGT	*#data*	0000000000 010010	Tcc2	19	WL
FTPLE	*#data*	0000000000 010101	Tcc2	19	WL
FTPLT	*#data*	0000000000 010100	Tcc2	19	WL
FTPNE	*#data*	0000000000 001110	Tcc2	19	WL
FTPNGE	*#data*	0000000000 011100	Tcc2	19	WL
FTPNGL	*#data*	0000000000 011001	Tcc2	19	WL
FTPNGLE	*#data*	0000000000 011000	Tcc2	19	WL
FTPNGT	*#data*	0000000000 011101	Tcc2	19	WL
FTPNLE	*#data*	0000000000 011010	Tcc2	19	WL
FTPNLT	*#data*	0000000000 011011	Tcc2	19	WL
FTPOGE	*#data*	0000000000 000011	Tcc2	19	WL
FTPOGL	*#data*	0000000000 000110	Tcc2	19	WL

FTPOGT	*#data*	0000000000 000010	Tcc2	19	WL
FTPOLE	*#data*	0000000000 000101	Tcc2	19	WL
FTPOLT	*#data*	0000000000 000100	Tcc2	19	WL
FTPOR	*#data*	0000000000 000111	Tcc2	19	WL
FTPSEQ	*#data*	0000000000 010001	Tcc2	19	WL
FTPSF	*#data*	0000000000 010000	Tcc2	19	WL
FTPSNE	*#data*	0000000000 011110	Tcc2	19	WL
FTPST	*#data*	0000000000 011111	Tcc2	19	WL
FTPT	*#data*	0000000000 001111	Tcc2	19	WL
FTPUEQ	*#data*	0000000000 001001	Tcc2	19	WL
FTPUGE	*#data*	0000000000 001011	Tcc2	19	WL
FTPUGT	*#data*	0000000000 001010	Tcc2	19	WL
FTPULE	*#data*	0000000000 001101	Tcc2	19	WL
FTPULT	*#data*	0000000000 001100	Tcc2	19	WL
FTPUN	*#data*	0000000000 001000	Tcc2	19	WL
FTWOTOX	FP*n*	000 000 000 0010001	Gen1	1	X
FTWOTOX	FP*n*,FP*n*	000 000 000 0010001	Gen1	2	X
FTWOTOX	<ea6>,FP*n*	010 000 000 0010001	Gen2	3	BWLSDXK

Table G-9
MC68851 Instructions

Opcode	Operands	Opcode Word	cp-Type	Group	Flags
PBAS	*label*	1111 000 01 0 000110	Bcc	4	
PBAC	*label*	1111 000 01 0 000111	Bcc	4	
PBBS	*label*	1111 000 01 0 000000	Bcc	4	
PBBC	*label*	1111 000 01 0 000001	Bcc	4	
PBCS	*label*	1111 000 01 0 001110	Bcc	4	
PBCC	*label*	1111 000 01 0 001111	Bcc	4	
PBGS	*label*	1111 000 01 0 001100	Bcc	4	
PBGC	*label*	1111 000 01 0 001101	Bcc	4	
PBIS	*label*	1111 000 01 0 001010	Bcc	4	
PBIC	*label*	1111 000 01 0 001011	Bcc	4	
PBLS	*label*	1111 000 01 0 000010	Bcc	4	
PBLC	*label*	1111 000 01 0 000011	Bcc	4	
PBSS	*label*	1111 000 01 0 000100	Bcc	4	
PBSC	*label*	1111 000 01 0 000101	Bcc	4	
PBWS	*label*	1111 000 01 0 001000	Bcc	4	
PBWC	*label*	1111 000 01 0 001001	Bcc	4	
PDBAS	*Dn,label*	0000000000 000110	DBcc	5	
PDBAC	*Dn,label*	0000000000 000111	DBcc	5	
PDBBS	*Dn,label*	0000000000 000000	DBcc	5	
PDBBC	*Dn,label*	0000000000 000001	DBcc	5	
PDBCS	*Dn,label*	0000000000 001110	DBcc	5	
PDBCC	*Dn,label*	0000000000 001111	DBcc	5	
PDBGS	*Dn,label*	0000000000 001100	DBcc	5	
PDBGC	*Dn,label*	0000000000 001101	DBcc	5	
PDBIS	*Dn,label*	0000000000 001010	DBcc	5	
PDBIC	*Dn,label*	0000000000 001011	DBcc	5	
PDBLS	*Dn,label*	0000000000 000010	DBcc	5	
PDBLC	*Dn,label*	0000000000 000011	DBcc	5	
PDBSS	*Dn,label*	0000000000 000100	DBcc	5	
PDBSC	*Dn,label*	0000000000 000101	DBcc	5	
PDBWS	*Dn,label*	0000000000 001000	DBcc	5	
PDBWC	*Dn,label*	0000000000 001001	DBcc	5	
PFLUSH	*special*	001 100 0 0000 00000		10	
PFLUSHA		001 001 0 0000 00000	Gen1	0	
PFLUSHR	*<ea12>*	101 0000000000000	Gen2	2	D
PFLUSHS	*special*	001 101 0 0000 00000		10	
PLOADR	*special*	001 000 1 0000 00000		11	
PLOADW	*special*	001 000 0 0000 00000		11	
PMOVE	*<ea0>*,BAD*n*	011 000 0 0000000 00	Gen2	12	W
PMOVE	BAD*n*,*<ea1>*	011 000 1 0000000 00	Gen3	12	W
PMOVE	*<ea0>*,BAC*n*	011 000 0 0000000 00	Gen2	12	W

PMOVE	BAC*n*,<*ea1*>	011 000 1 0000000 00	Gen3	12	W
PMOVE	<*ea0*>,PSR	011 000 0 0000000 00	Gen2	12	W
PMOVE	PSR,<*ea1*>	011 000 1 0000000 00	Gen3	12	W
PMOVE	<*ea0*>,TC	010 000 0 000000000	Gen2	12	L
PMOVE	TC,<*ea1*>	010 000 1 000000000	Gen3	12	L
PMOVE	<*ea0*>,*x*RP	010 000 0 000000000	Gen2	12	D
PMOVE	*x*RP,<*ea1*>	010 000 1 000000000	Gen3	12	D
PMOVE	<*ea0*>,SCCCAL	010 000 0 000000000	Gen2	12	B
PMOVE	SCCCAL,<*ea1*>	010 000 1 000000000	Gen3	12	B
PMOVE	<*ea0*>,VAL	010 000 0 000000000	Gen2	12	B
PMOVE	VAL,<*ea1*>	010 000 1 000000000	Gen3	12	B
PMOVE	<*ea0*>,AC	010 000 0 000000000	Gen2	12	W
PMOVE	AC,<*ea1*>	010 000 1 000000000	Gen3	12	W
PMOVE	PCSR,<*ea1*>	011 000 1 0000000 00	Gen3	12	W
PRESTORE	<*ea8*>	1111 000 101 000 000	Rest	6	
PSAVE	<*ea4*>	1111 000 100 000 000	Save	6	
PSAS	<*ea3*>	0000000000 000110	Scc	7	B
PSAC	<*ea3*>	0000000000 000111	Scc	7	B
PSBS	<*ea3*>	0000000000 000000	Scc	7	B
PSBC	<*ea3*>	0000000000 000001	Scc	7	B
PSCS	<*ea3*>	0000000000 001110	Scc	7	B
PSCC	<*ea3*>	0000000000 001111	Scc	7	B
PSGS	<*ea3*>	0000000000 001100	Scc	7	B
PSGC	<*ea3*>	0000000000 001101	Scc	7	B
PSIS	<*ea3*>	0000000000 001010	Scc	7	B
PSIC	<*ea3*>	0000000000 001011	Scc	7	B
PSLS	<*ea3*>	0000000000 000010	Scc	7	B
PSLC	<*ea3*>	0000000000 000011	Scc	7	B
PSSS	<*ea3*>	0000000000 000100	Scc	7	B
PSSC	<*ea3*>	0000000000 000101	Scc	7	B
PSWS	<*ea3*>	0000000000 001000	Scc	7	B
PSWC	<*ea3*>	0000000000 001001	Scc	7	B
PTESTR	*special*	100 000 1 0000 00000		13	
PTESTW	*special*	100 000 0 0000 00000		13	
PTAS		0000000000 000110	Tcc1	8	
PTAC		0000000000 000111	Tcc1	8	
PTBS		0000000000 000000	Tcc1	8	
PTBC		0000000000 000001	Tcc1	8	
PTCS		0000000000 001110	Tcc1	8	
PTCC		0000000000 001111	Tcc1	8	
PTGS		0000000000 001100	Tcc1	8	
PTGC		0000000000 001101	Tcc1	8	
PTIS		0000000000 001010	Tcc1	8	
PTIC		0000000000 001011	Tcc1	8	
PTLS		0000000000 000010	Tcc1	8	

PTLC		0000000000 000011	Tcc1	8	
PTSS		0000000000 000100	Tcc1	8	
PTSC		0000000000 000101	Tcc1	8	
PTWS		0000000000 001000	Tcc1	8	
PTWC		0000000000 001001	Tcc1	8	
PTPAS	*#data*	0000000000 000110	Tcc2	9	WL
PTPAC	*#data*	0000000000 000111	Tcc2	9	WL
PTPBS	*#data*	0000000000 000000	Tcc2	9	WL
PTPBC	*#data*	0000000000 000001	Tcc2	9	WL
PTPCS	*#data*	0000000000 001110	Tcc2	9	WL
PTPCC	*#data*	0000000000 001111	Tcc2	9	WL
PTPGS	*#data*	0000000000 001100	Tcc2	9	WL
PTPGC	*#data*	0000000000 001101	Tcc2	9	WL
PTPIS	*#data*	0000000000 001010	Tcc2	9	WL
PTPIC	*#data*	0000000000 001011	Tcc2	9	WL
PTPLS	*#data*	0000000000 000010	Tcc2	9	WL
PTPLC	*#data*	0000000000 000011	Tcc2	9	WL
PTPSS	*#data*	0000000000 000100	Tcc2	9	WL
PTPSC	*#data*	0000000000 000101	Tcc2	9	WL
PTPWS	*#data*	0000000000 001000	Tcc2	9	WL
PTPWC	*#data*	0000000000 001001	Tcc2	9	WL
PVALID	VAL,*<ea11>*	001 010 0000000000	Gen3	3	L
PVALID	A*n*,*<ea11>*	001 011 0000000000	Gen3	3	L

Appendix H

Assembler Command Syntax

Syntax Asm [*option* ...] [*file* ...]

Description Assembles the specified assembly-language source files. One or more filenames may be specified. If no filenames are specified, standard input is assembled and the file "a.o" is created. By convention, assembly-language source file names end in the suffix ".a". Each file is assembled separately— assembling file *name*.a creates object file *name*.a.o. The object file name can be changed with the **-o** option.

See the *MPW Assembler Reference* manual for details of the assembly language.

Input If no filenames are specified, standard input is assembled. (End-of-file is indicated by typing Command-Enter.)

Output If either the **-l** or **-s** option is specified, an assembler listing is generated. If standard input is used for the source file, the listing is written to standard output. If the input is taken from file *name*.a, the listing is written to *name*.a.lst. The listing file name can be changed with the **-lo** option.

Diagnostics Errors and warnings are written to diagnostic output. If the **-p** option is specified, progress and summary information is also written to diagnostic output.

Status

The following status values are returned to the Shell:

0 No errors detected in any of the files assembled
1 Parameter or option errors
2 Errors detected
3 Execution terminated

Options

Except for the **-case on** option, options may appear in any order.

-addrsize *size* Set address displays in the listing to *size* digits (values 4 to 8 are allowed). The default is 5 digits.

-blksize *blocks* Set the Assembler's text file I/O buffer size to *blocks**512 bytes. Values 6 to 62 are allowed. Odd values are made even by reducing the value by 1. The default value is 16 (8192 bytes) if the Assembler determines it has the memory space for the I/O buffers, and 6 (3072 bytes) otherwise. This option permits optimization of I/O performance (transfer rate for text file input, load/dump files, and listing output) as a function fo the disk device being used. Note that increasing the blocks value reduces the amount of memory available for other Assembler structures (such as symbol tables).

-case on Distinguish between upper- and lowercase letters in non-macro names (same as CASE ON). (Case is always ignored in macro names.) If you intend to preserve the case of names declared by the **-define** option, then the **-case on** option must *precede* the **-define** option(s) in the command parameter list.

-case obj[ect] Preserve the case of module, EXPORT, IMPORT, and ENTRY names *only in the generated object file.* In all other respects, case is ignored within the assembly, and the behavior is the same as the preset CASE OFF situation.

-case off Ignore the case of letters. All identifiers are case insensitive. This is the preset mode of the Assembler, but it may be used in the command line to reverse the effect of one of the other **-case** modes.

-c[heck] Syntax check only. No object file is generated.

-d[efine] *name*[=*value*] [,*name*[=*value*]]...
Define the name as having the specified value. The value is a decimal integer. If *value* is omitted, a value of 1 is assumed. This option is equivalent to placing the directive

```
name EQU value
```

at the beginning of your source file. Note that in order to test whether or not the name is defined, the **&Type** function should be used. You can define more than one name by specifying multiple **-d** options or multiple *name*[=*value*] parameters separated by commas. For example,

```
Asm -d debug1,&debug='on'  ...
```

-d[**efine**] **&***name*[=[*value*]] [,**&***name*[=[*value*]]]...

Define the macro *name* as having the specified value. The value is a decimal integer or a string constant. If the "=*value*" is omitted, the decimal value 1 is assumed. If only the *value* is omitted, the null string is assumed. **-define** is equivalent to declaring the name as a global arithmetic symbol (GBLA for an integer value) or global character macro symbol (GBLC for a string value) and placing one of the following directives at the beginning of the source file:

Note that in order to test whether the name is defined, the **&Type** function should be used. You can define more than one macro name by specifying multiple **-d** options or multiple &*name*[=*value*] parameters separated by commas.

-e[**rrlog**] *filename*

Write all errors and warnings to the error log file with the specified filename (same as ERRLOG '*filename*').

-f

Suppress page ejects (same as PRINT NOPAGE).

-font *fontname*[,*fontsize*]

Set the listing font to *fontname* (for example, Courier), and the size to *fontsize*. This option is meaningful only if the **-s** or the **-l** option is used. The default listing font is Monaco 7. Note that listings will be formatted correctly only if a monospaced font is used.

-h

Suppress page headers (same as PRINT NOHDR).

-i *pathname* [,*pathname*]...

Search for include and load files in the specified directories. Multiple **-i** options may be specified. At most 15 directories will be searched. The search order is as follows:

1. The include or load filename is used as specified. If a *full pathname* is given, then no other searching is applied.

   If the file wasn't found, and the pathname used to specify the file was a *partial pathname* (no colons in the name or a leading colon), then the following directories are searched.

2. The directory containing the current input file.

3. The directories specified in **-i** options, in the order listed.

4. The directories specified in the Shell variable {AIncludes}.

-l Generate full listing. If file *name*.a is assembled, the listing is written to *name*.a.lst.

-lo *listingname* Pathname for the listing file and directory for the listing scratch file. If *listingname* ends with a colon (:), it indicates a directory for the listing file, whose name is then formed by the normal rules (that is, *inputFilename*.a.lst). If *listingname* does not end with a colon, the listing file is written to the file *listingname*. In this case, listings for multiple source files are appended to the listing file. In either case, the directory implied by the listing name is used for the assembler's listing scratch file. The **-lo** option is only meaningful if the **-s** or the **-l** option is used.

-o *objname* Pathname for the generated object file. If *objname* ends with a colon (:), it indicates a directory for the output file, whose name is then formed by the normal rules (that is, *inputFilename*.o). If *objname* does not end with a colon, the object file is written to the file *objname*. (In this case, only one source file should be specified to the Assembler.)

-pagesize *[l,w]* Set the listing page size. (This option is only meaningful if the **-s** or **-l** option is specified.) The *l* and *w* parameters are integers: *l* is the page length (default = 75) and *w* is the page width (default = 126). (These settings assume that Monaco 7 is being used with the MPW Print command to the LaserWriter.)

-print *mode* [,*mode*]...

Set a print option mode. *Mode* may be any one of the following PRINT directive options:

[NO]GEN	macro expansions
[NO]PAGE	page ejects
[NO]WARN	warnings
[NO]MCALL	macro calls
[NO]OBJ	object code
[NO]DATA	data
[NO]MDIR	macro directives
[NO]HDR	page headings
[NO]LITS	literals
[NO]STAT	progress information
[NO]SYM	symbol table display

See the *MPW Assembler Reference* manual for a discussion of these PRINT settings. You can specify more then one print option by specifying multiple **-print** options or multiple *mode* parameters separated by commas. For example,

```
Asm -print nowarn,noobj,nopage  ...
```

Note that single-letter options are provided for some of the settings: **-f** (NOPAGE), **-h** (NOHDR), **-p** (STAT), and **-w** (NOWARN).

-p Write assembly progress information (module names, included, loads, and dumps) and summary information (number of errors, warnings, and compilation time) to the diagnostic output file. (This option is the same as PRINT STAT.)

-s Set PRINT NOOBJ to generate a shortened form of the listing file. If the **-l** option is also specified, the rightmost option takes precedence.

-t Display the assembly time and the number of lines to the diagnostic file even if progress information (**-p**) is not being displayed.

-w Suppress warning messages (same as PRINT NOWARN).

-wb Suppress branch warning messages only.

Examples

```
Asm -w -l Sample.a Memory.a -d Debug
```

Assembles Sample.a and Memory.a, producing object files Sample.a.o and Memory.a.o. Suppresses warnings and define the name "Debug" as having the value 1. Two listing files are generated: Sample.a.lst and Memory.a.lst. (These programs are located in the AExamples directory.)

Appendix I

Programming for the Shell Environment

The MPW libraries contain four groups of utility routines for assembly-language tools that run within the MPW Shell environment. They are:

- Shell environment routines: procedures, functions, and data structures required to access MPW command-line parameters, Shell variables, and the standard diagnostic file
- Shell signal handling routines: procedures and functions that give you access to MPW software interrupts
- MPW cursor control routines: procedures that let you control the form and action of the cursor
- error message file management routines: procedures that let you access messages in the Macintosh system error message file

If you are writing a tool, you may want to use some or all of these routines. General guidelines for writing MPW tools are given in *Macintosh Programmer's Workshop Reference*.

General requirements

To use any of the routines described in this appendix, you must do the following:

- import the names of the routines you are using
- use the correct calling conventions

- call the `_RTInit` function early in your program and the `exit` or `_exit` procedure at the end
- link your assembly with the library or libraries that contain the routines' code

These requirements are discussed below.

Importing the routines

Import the names of the routines described in this appendix by using `IMPORT` directives. For the Shell environment and signal handling routines, you can simply include the files IntEnv.a and/or Signal.a; they contain the required directives. For the cursor control and error file management routines, you must write your own `IMPORT` directives in your source text.

The Shell environment and signal handling routines are mostly C routines; hence their names are case-sensitive. The cursor control and error file management routines are all Pascal routines. Their names are not case-sensitive unless `CASE OBJ` or `CASE ON` is in effect, in which case their names must be imported in capital letters.

Calling conventions

The routines described in this appendix use both Pascal and C calling conventions. In all cases,however, their call formats are given in Pascal. Those that use C calling conventions are flagged by a `C` directive at the end of the Pascal call. Both Pascal and C calling conventions are explained in Appendix L.

As a convenience, the parameters to be pushed on the stack are listed, in the order they are pushed, below the Pascal call format. A right-pointing arrow before the parameter name indicates a value passed to the routine; a left-pointing arrow indicates a value returned by the routine.

All functions described in this appendix leave their results in register D0.

The _RTInit function

```
FUNCTION _RTInit (retPC: ptr; VAR pArgC, pArgV, pEnvP: longint; forPascal:
                  boolean): boolean; C;

-->     forPascal       0 for C strings, 1 for Pascal strings
<--     pEnvP           the address of EnvP, value set by _RTInit
<--     pArgV           the address of ArgV, value set by _RTInit
<--     pArgC           the address of Argc, value set by _RTInit
-->     retPC           main program return address
```

One of the first calls in your program must be to the `_RTInit` function; the very last call should be to the `exit` or `_exit` procedure, which calls the `_RTExit` procedure. `_RTInit` is described in this section; `_RTExit` is described later in this appendix under "Shell Utility Routines."

The `_RTInit` function allocates approximately 500 bytes of nonrelocatable space in the heap and calls `_DataInit`, the routine that initializes global data. `_RTInit` must be called before any of the other routines described in this appendix; if possible, it should be called before other code segments have been loaded.

The `_RTInit` function has these parameters:

- *forPascal* is a numeric value passed to `_RTInit`. Its value should be zero if you want the strings pointed to by *pEnvP* and *pArgV* to be in C format (terminated by a zero character), and one if you want them to be in Pascal format (preceded by a length byte).
- *pEnvP* points to a pointer variable that `_RTInit` will set to the vector of exported Shell variables.
- *pArgV* points to a pointer variable that `_RTInit` will set to the value of the Shell variable `ArgV`. `ArgV` is discussed below, under "Accessing MPW Command-Line Parameters."
- *pArgC* points to a long integer that `_RTInit` will set to the value of the Shell variable `ArgC`. `ArgC` is discussed below, under "Accessing MPW Command-Line Parameters."
- *retPC* is the address to which program control should pass upon execution of `_RTExit`, as described later in this appendix under "Shell Utility Routines."

The `_RTInit` function returns a value of one if your program is being launched by the Macintosh Finder, and zero if it is being launched by the MPW Shell. This is the value placed in the `StandAlone` variable, described below under "Shell Utility Routines."

Files to link with

The code for the Shell environment and signal handling routines is in the library Runtime.o, except for the code for the `IEGetEnv` function, which is in PasLib.o. The code for the cursor control and error file management routines is in the library ToolLibs.o. You must link the appropriate file or files to your assembly if you use any of these routines.

Shell Environment routines

To access the MPW Shell Environment, do the following:

- Include the file IntEnv.a in your source text.
- Link your program with the file Runtime.o. If your program calls `IEGetEnv`, link it also with the file PasLib.o.

Accessing MPW command-line parameters

This section describes the data structures that allow you to access the information given in an MPW command line.

The ArgC variable

The value of the variable `ArgC`, set by `__RTInit`, is determined by the number of parameters supplied to the program on the command line. The value of `ArgC` is always greater than or equal to one, because the first "parameter" is always the command name. For example, the command

```
catenate newfile
```

leaves a value of 2 in the `ArgC` variable.

How Shell commands are stored and accessed

The `ArgV` variable, set by `_RTInit`, is a pointer to an array of type `ARRAY[0..ArgC]` of pointers, dynamically allocated and initialized by the MPW Shell when a program begins execution. Each command-line parameter to the program is stored as a Pascal-formatted or C-formatted string (depending on the value of the *forPascal* parameter passed to `_RTInit`), pointed to by a pointer in the array.

Element 0 of ArgV is always the command name, as supplied by the user. Any time the value of `ArgC` is greater than one, which indicates that there are parameters associated with the command given, the parameters are stored in `ArgV^[1]^` (the first string after the command name) through `ArgV^[ArgC-1]^` (because the first element in the array is 0). The value of `ArgV^[ArgC]` always has a value of zero (`nil`).

Shell utility routines

A variable and six utility routines, described in this section, are available for use with programs that run within the MPW environment.

The StandAlone variable

The `StandAlone` variable contains a value of zero if the current program is running under the MPW Shell, and one if the current program is running standalone.

The getenv function

```
getenv (VAR varname: Cstring): Cstrptr; C;

->      varname         the address of the C-string variable name to find
```

The `getenv` function searches for the value of a variable that is defined and exported in the MPW Shell environment. *Macintosh Programmer's Workshop Reference* contains a list of standard exported Shell variables. The parameter *varname* is a pointer to a C-formatted string naming an exported Shell variable, with uppercase and lowercase not distinguished. If `getenv` is successful in finding the variable, it leaves a pointer to the variable's C-formatted string value in register D0; otherwise it sets D0 to zero.

❖ *Note:* Use `getenv` only if the value of the *forPascal* parameter in the `_RTInit` call was zero; use `IEGetEnv` otherwise.

The IEGetEnv function

```
IEGetEnv(varValue, varName: strptr): boolean;

<-      varValue        address of the Pascal string value for TRUE result
->      varName         address of the Pascal string variable name to find
```

`IEGetEnv` is a function that returns a value of one if it is successful in finding the value of a variable that is defined and exported in the MPW Shell environment. *Macintosh Programmer's Workshop Reference* contains a list of standard exported Shell variables. The parameter *varName* is a pointer to a Pascal string naming an exported Shell variable, with uppercase and lowercase not distinguished. The parameter *varValue* is returned with a pointer to the value of the Shell variable. The `IEGetEnv` function returns zero if it cannot find the variable.

❖ *Note:* Use `IEGetenv` only if the value of the *forPascal* parameter in the `_RTInit` call was one; otherwise use `getenv`.

The onexit procedure

```
onexit(exitProc: ptr); C;

->      exitProc        the address of a routine to call when exit is called
```

Normal program termination closes and flushes open files and releases program memory. If you want additional exit processing, you can use `onexit` to insert a routine that is executed just before normal termination. The parameter *exitProc* is a pointer to such a routine, which has no parameters. Multiple calls to `onexit` are permitted, up to a maximum of six. The routines specified will be executed in the reverse order of their installation when `_RTExit` is called. A call to `_exit` does not call the routines installed by `onexit`.

The exit and _exit procedures

```
exit(status: longint); C;

_exit(status: longint); C;

->      status          the value to return to the MPW shell
```

Both the `exit` and `_exit` procedures terminate a program running under the MPW Shell by masking out the upper byte of *status* and calling `_RTExit`. The action of _RTExit is described below.

The `exit` procedure calls all procedures installed by `onexit` and flushes all files before closing them; `_exit` does not. Use `_exit` if you are uncertain about the integrity of your program's data space.

The _RTExit procedure

```
_RTExit(status: longint);

->      status          pass to shell variable {status}
```

The `_RTExit` procedure must be the last executed routine in a tool running under the MPW Shell. It calls any routines installed by the `onexit` procedure (described above) and then returns control to the address specified by the *retPC* parameter in the original `_RTInit` call, described under "General Requirements." Open files are closed and all memory allocated to the program is released. This procedure does not close files opened by calls to the Macintosh ROM routines, however; the Shell closes them after the program returns. The parameter *status* is passed to the Shell variable `{status}`; it should be 0 for successful execution, and a 24-bit nonzero value for a termination error. The upper byte of *status* is reserved for use by the signal handler (described later in this appendix).

Programs normally call the `exit` or `_exit` procedure, described above. These routines call `_RTExit` after masking off the upper byte of *status*.

Shell I/O routines

Eight general I/O routines are available for use with programs that run within the MPW environment.

Error information

All Shell I/O routines report errors by setting the value of the integer variable `errno`. In addition, the routines open, close, read, write, and ioctl set the Macintosh system variable `MacOSErr`. The error values are shown in Table I-1.

Table I-1
Shell I/O errors

Value	Identifier	Message	Explanation
2	ENOENT	No such file or directory	This error occurs when a file whose filename is specified does not exist or when one of the directories in a pathname does not exist.
3	ENORSRC	Resource not found	A required resource was not found. This error applies to `faccess` calls that return tab, font, or print record information.
5	EIO	I/O error	Some physical I/O error has occurred. This error may in some cases be signaled on a call following the one to which it actually applies.
6	ENXIO	No such device or address	I/O on a special file refers to a subdevice that does not exist, or the I/O is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.
7	E2BIG	Insufficient space for return argument	The data to be returned is too large for the space allocated to receive it.
9	EBADF	Bad file number	Either a file descriptor does not refer to an open file, or a read (or write) request is made to a file that is open only for writing (or reading).
12	ENOMEM	Not enough space	The system ran out of memory while the library call was executing.
13	EACCES	Permission denied	An attempt was made to access a file in a way forbidden by the protection system.
17	EEXIST	File exists	An existing file was mentioned in an inappropriate context.
19	ENODEV	No such device	An attempt was made to apply an inappropriate system call to a device; for example, read a write-only device.
20	ENOTDIR	Not a directory	An object that is not a directory was specified where a directory is required; for example, in a path prefix.
21	EISDIR	Is a directory	An attempt was made to write on a directory.

22	EINVAL	Invalid parameter	Some invalid parameter was provided to a library function.
23	ENFILE	File table overflow	The system's table of open files is full, so temporarily a call to open cannot be accepted.
24	EMFILE	Too many open files	The system cannot allocate memory to record another open file.
28	ENOSPC	No space left on device	During a write to an ordinary file, there is no free space left on the device.
29	ESPIPE	Illegal seek	An `lseek` was issued incorrectly.
30	EROFS	Read-only file system	An attempt to modify a file or directory was made on a device mounted for read-only access.
31	EMLINK	Too many links	An attempt to delete an open file was made.

The faccess function

```
faccess(filename: strptr; cmd, arg: longint): longint; C;

-->     arg             an optional argument, depending on the value of cmd
-->     cmd             the value or address of the argument
-->     filename        the address of the filename
```

The `faccess` function provides access to control and status information for named files. The parameter *filename* is a pointer to a C-formatted string containing the name of an existing file. The parameter *cmd* contains one of the following constants, declared in the include file IntEnv.a:

F_DELETE	Deletes the named file, or returns an error if the file is open or in a window. *Arg* is ignored.
F_RENAME	Renames the named file. *Arg* is a pointer to a C-formatted string containing the new name.
F_GTABINFO	Returns the tab setting for an MPW text file named by *filename*. *Arg* is a pointer to a long integer. The long integer's value is the tab setting expressed as the number of spaces in the text file's font.
F_STABINFO	Sets the tab setting for an MPW text file named by *filename*. *Arg* is a long integer representing the tab setting expressed as the number of spaces in its font.

F_GFONTINFO Returns the font and font size of an MPW text file named by *filename*. *Arg* is a pointer to a long integer. The font number is stored in the upper word of the long integer; the font size is stored in the lower word.

F_SFONTINFO Sets the font and font size of an MPW text file named by *filename*. *Arg* is a long integer. The font number is read from the upper word of the long integer; the font size is read from the lower word.

F_GPRINTREC Gets a print record `TPrint` for the MPW text file *filename*. *Arg* is a handle to the print record. Before calling `faccess` with this *cmd* value, the Macintosh Printing Manager must be initialized and the print record handle `THPrint` must be allocated.

F_SPRINTREC Sets a print record for the MPW text file *filename*. *Arg* is a handle to the print record. Before calling `faccess` with this *cmd* value, the Macintosh Printing Manager must be initialized and the print record handle `THPrint` must be allocated.

F_OPEN Reserved for operating system use.

❖ *Note:* `faccess` with *cmd* values of `F_DELETE` and `F_RENAME` can be used by all programs. `faccess` with other values of *cmd* can be used only by tools running within the MPW environment.

If `faccess` is successful it returns a nonnegative value, usually 0, in D0. If the file cannot be accessed, `faccess` returns –1 and sets the value of `errno`.

The open procedure

```
open(filename: strptr; mode: longint); C;

->      mode            the value of the mode of opening
->      filename        the address of the filename to open
```

The `open` procedure opens the file, window, or selection named by the C-formatted string *filename* for both reading and writing. It can be used only with untyped file variables.

The parameter *mode* can specify automatic file creation, truncation, and/or exclusive access. To construct a value for *mode*, combine the following constants (declared in the include file IntEnv.a) by combining their values with `OR`:

O_RDONLY Open the file for reading only.

O_WRONLY	Open the file for writing only.
O_RDWR	Open the file for both reading and writing.
O_APPEND	Set the file pointer to the end of the file before each write.
O_CREAT	If the file does not currently exist, create it.
O_TRUNC	If the file currently exists, truncate its length to zero. Its *mode* setting and owner will remain unchanged.
O_RSRC	Open the file's resource fork, instead of its data fork.
O_EXCL	Cancel open if the file currently exists. This *mode* value is valid only if O_CREAT is also specified.

After open executes successfully, D0 contains an integer file descriptor, with the file pointer set to the beginning of the file. If there is an error, D0 will contain –1 and errno will be set. File descriptors for input, output, and the diagnostic output are predeclared in the include file IntEnv.a, as shown in Table I-2.

Table I-2
Predeclared file descriptors

Value	Identifier	File
0	InputFD	Standard input
1	OutputFD	Standard output
2	DiagnosticFD	Diagnostic output

The close function

```
close(fd: longint): longint; C;
->      fd              the value of the file descriptor
```

The close function closes the file associated with the file descriptor *fd*. If it is successfull, close sets D0 to zero; otherwise it sets it to a nonzero value and sets a value in errno.

The read function

```
read(fd: longint; char: bufptr; count: longint): longint; C;

->      count           the value of the number of characters to read
->      char            the address of the buffer in which to read
->      fd              the value of the file descriptor to read
```

The `read` function transfers up to *count* bytes from the file associated with *fd* into the buffer pointed to by *char*. The transfer begins from the current file position. Upon completion, `read` increments the current file position by the number of bytes actually read.

If successful, `read` leaves the number of bytes actually read in D0 (which may be less than *count*, if the end-of-file was encountered); otherwise it sets D0 to –1 and sets the value of `errno`.

The write function

```
write(fd: longint; char: bufptr; count: longint): longint; C;

->      count           the value of the number of characters to write
->      char            the address of the buffer from which to write
->      fd              the value of the file descriptor to write to
```

The `write` function transfers up to *count* bytes from the buffer pointed to by *char* into the file associated with *fd*. The transfer begins from the current file position (unless the file was opened with `O_APPEND`). Upon completion, `write` increments the current file position by the number of bytes actually written.

If successful, `write` leaves the number of bytes actually written in D0; otherwise it sets D0 to –1 and sets the value of `errno`.

The ioctl function

```
ioctl(fd, cmd: longint): longint; C;

->      cmd             the control/status selector
->      fd              the value of the file descriptor
```

The `ioctl` function communicates with a file's device handler by sending control information and/or requesting status information. The *fd* parameter is a file descriptor. Predeclared file descriptor values are shown in Table I-2.

The *cmd* parameter specifies one of the following device-specific operations, using constants declared in the include file IntEnv.a:

`FIOINTERACTIVE`	Return a value of 0 if the device is interactive, –1 otherwise. Ignore *arg*.
`FIOBUFSIZE`	Return the default buffer size for the device. The buffer size is expressed in bytes and is returned as a `longint` value pointed to by *arg*. If the device has no default buffer size, `ioctl` returns a value of –1; it returns 0 otherwise.
`FIOFNAME`	Store the filename associated with *fd* in a character array 255 characters in size,pointed to by *arg*. `ioctl` returns a value of –1 if the filename length exceeds 255 characters, 0 otherwise.
`FIOREFNUM`	Return the Macintosh file reference number associated with *fd*. The reference number is returned as an `integer` value pointed to by *arg*. `ioctl` returns a value of –1 if the file associated with *fd* is not open on a Macintosh file (such as the console device), 0 otherwise. This value lets you do Macintosh I/O operations (such as `Allocate`) that are not otherwise available through `ioctl`.
`FIOSETEOF`	Set the logical end of the file associated with *fd* to the value of *arg* , which becomes the new size of the file in bytes. This command can be used to reduce or increase the size of an open file. The current file pointer is not affected unless the file size is set to a value lower than the position to which it points.
`TIOFLUSH`	Discard unread terminal input. This parameter value is used only for the console device and other terminal devices. `ioctl` returns a value of –1 if the file associated with *fd* is not a terminal device, 0 otherwise. Parameter *arg* is ignored.
`TIOGPORT`	Return the console's GrafPort by means of a `GrafPtr` pointed to by *arg*. This parameter value is used only for the console device when running standalone. If you use this value to set the console's GrafPort, do not deallocate the GrafPort space; the console will be written to when your program exits.
`TIOSPORT`	Set the console's GrafPort by means of a `GrafPtr` pointed to by *arg*. Subsequent writes to the console will display on this GrafPort. This parameter value is used only for the console device when running standalone. It lets you direct debugging output from your program to a specified GrafPort.

The lseek function

```
lseek(fd, bytePos, mode: longint): longint; C;

->      mode            the value of the mode selector
->      bytePos         the value of the postion to seek
->      fd              the value of the file descriptor
```

moves the read/write file pointer in the file associated with *fd*, according to the following values of *mode* and *bytePos*:

- If the value of *mode* is `fromStart` (0), it sets the pointer to *bytePos* bytes. The value of *bytePos* must be positive.
- If the value of *mode* is `fromCurPos` (1), it adds *bytePos* to the current pointer position. The value of *bytePos* may be negative.
- If the value of *mode* is `fromEnd` (2), it sets the pointer to the current file size plus *bytePos*. The value of *bytePos* may be negative.

The *mode* values given above are declared in the include file IntEnv.a. Predeclared file descriptors are shown in Table I-2.

After successful execution, `lseek` returns the file pointer value in bytes from the beginning of the file. The pointer value remains unchanged if the file associated with *fd* is not open or if the result of `lseek` would position the pointer before the beginning of the file or after its end.

The `lseek` function has no effect on a file opened with `open` and the `O_APPEND` mode, because the next writing action repositions the file pointer to the end of the file.

The fcntl function

```
fcntl(fd, cmd, lowestFD: longint): longint; C;

->      lowestFD        lowest limit of new FD value
->      cmd             the constant F_DUPFD (value = 0)
->      fd              the value of the file descriptor
```

The `fcntl` function returns in D0 the value of an open file descriptor that duplicates the file designated by *fd* and shares the same read/write pointer. This value will be the lowest available file descriptor greater than *lowestFD*. The parameter *cmd* must be set to `F_DUPFD` (zero).

Shell signal handling routines

Tools that run under MPW can respond to software interrupts—known as *signals*—by using the signal-handling routines described in this section.

To access the signal handler, do the following:

- Include the file Signal.a in your source text.
- Link your program with the file Runtime.o.

A signal is similar to a hardware interrupt in that it can cause program control to be diverted from its normal execution sequence. The difference is that the events that raise a signal reflect a change in program state rather than hardware state. Examples of signal events are stack overflow, heap overflow, software floating-point errors, and Command-period interrupts.

❖ *Note:* The only software interrupt that can be detected by a program running under the current version of the MPW Shell is the Command-period, which is represented by the value `sigInt`. As additional software interrupts are added new values will be added to represent them, which the signal-handling procedures will then accept.

Signals can be caught, held and then released, or ignored. The default action of any signal raised is to close all open files, execute any exit procedures installed with the `onexit` procedure, and terminate the program. No signal-handling calls are required to execute a normal termination when a signal is detected.

If your program requires special handling of a signal or chooses to ignore it, the `sigset` function lets you replace the default procedure with one of your own. You can also temporarily "hold" (that is, suspend) action on a signal by calling the `sighold` procedure. You may want to do this before entering a critical section of code. The signal can then be restored by calling the `sigrelease` procedure, whereupon its signal-handling procedure will take effect if the signal was raised since the preceding call to `sighold`. Your program may also wait until one or more signals are raised by calling the `sigpause` procedure.

A signal is represented by a bit in the value `sigMap`. The signal-handling procedures accept a `sigMap` that can specify several signals as a group. You can refer to several signals at once by adding their values. You can refer to all signals at once by using the value `SIGALLSIGS`.

The sigset function

```
sigset(sigMap: longint; newHandler: ptr): ptr; C;

->      newHandler      the address of the routine to install
->      sigMap          a bitmap of the signals to be handled
```

The `sigset` function replaces the current signal handler (the procedure to be executed upon receipt of the signals specified in *sigMap*) with a user-supplied signal handler. The default signal handler may be set or restored by specifying `SIG_DFL` as the current signal handler. The signals may be ignored entirely by specifying `SIG_IGN` as the current signal handler.

The `sigset` function returns the previous `SignalHandler` pointer in D0. If this pointer must be restored in another part of the program, save the return value and restore it with another call to `sigset`. Multiple signals may be set with one call to `sigset` by combining signal values together with `OR` in *sigMap*, but in this case `sigset` cannot, of course, return all previous values and its return value is meaningless. To correctly save multiple previous signal handlers, call `sigset` separately for each signal.

The sighold function

```
sighold(sgMap: longint): longint; C;

->      sigMap          the value of the signals to suspend
```

The `sighold` function permits temporary suspension and restoration of signals. Before a program enters a critical section of code, it should call `sighold` with a signal map of signals to suspend or with the value `SIGALLSIGS`, which represents all signals. Function `sighold` returns a `SignalMap` in D0, representing the list of signals already being held; this value should be saved for use as the *prevEnabled* parameter in the subsequent call to `sigrelease`. If the signal event (such as Command-period) occurs after a call to `sighold` is made, the event is recorded in the signal state but the signal handler is not executed.

The sigrelease procedure

```
sigrelease(sgMap, prevMap: longint); C;

->      prevMap         the map of signals previously held
->      sigMap          the map of signals to restore
```

The `sigrelease` procedure lets you reenable signals that were held by a previous call to `sighold`, by specifying their corresponding bits in *sigMap*. Signals that were already on hold when you called `sighold` should be specified to `sigrelease` in the *prevEnabled* parameter to permit correct handling of nested calls to `sighold`. If any of the signal events occurred while they were held, their signal-handling routines will take effect immediately after the return from `sigrelease`. Signal events do not stack; multiple occurrences of signal events that are being held do not yield multiple invocations of the signal handler when the signal is released.

The sigpause procedure

```
sigpause(sgMap: SignalMap); C;

->      sigMap          the signals to release the program
```

A call to `sigpause` suspends program activity until a signal event is recorded for any signal not currently held. It is intended for signal synchronization, though in the current implementation its application is limited; it is included here in order to provide a complete signal environment model.

MPW cursor control routines

Four procedures in the MPW Tools library let you control the appearance and action of the MPW cursor. These routines all use Pascal calling conventions and Pascal-formatted strings.

To access the cursor control unit, link your assembly with the file ToolLibs.o.

The Show_Cursor procedure

```
Show_Cursor(CursorKind: SignedByte);

->      CursorKind      the cursor pattern to display
```

The `Show_Cursor` procedure increments the cursor level (which may have been decremented by `Hide_Cursor`). If the level is zero, it displays the cursor. The cursor level never increments above zero. The parameter *CursorKind* lets you select the form of the cursor, as shown in Table I-3.

Table I-3
Cursor kinds

Value	Cursor
0	Hidden cursor
1	I-beam
2	Cross
3	Plus sign
4	Watch
5	Arrow

A *CursorKind* value of `HIDDEN_CURSOR` causes only a Macintosh `ShowCursor` call; the other values precede the `Show_Cursor` call with a `SetCursor` call.

The Hide_Cursor procedure

```
Hide_Cursor;
```

The `Hide_Cursor` procedure simply calls the Macintosh `HideCursor` routine. For further information, see the QuickDraw chapter of *Inside Macintosh*.

The RotateCursor procedure

```
RotateCursor(Counter: Integer);

->      Counter         value to control cursor rotation
```

The `RotateCursor` procedure performs a Macintosh `SetCursor` call for the "beach ball" cursor form and rotates it one-quarter turn whenever the value of *Counter* `MOD 32` is zero. It assumes that the cursor is visible from a prior `Show_Cursor` call. To use `RotateCursor`, your program must set up and increment a suitable counter. If the value of *Counter* is positive, the cursor rotates clockwise; if it is negative, it rotates counterclockwise.

The SpinCursor procedure

```
SpinCursor(Increment: integer);

->      Increment       value to control rotation counter
```

The `SpinCursor` procedure performs the same actions as `RotateCursor`, but maintains its own internal counter. Your program specifies the *Increment* to be counted (either positive or negative), and `SpinCursor` adds it to its counter. A positive increment spins the cursor clockwise; a negative increment spins it counterclockwise. An *Increment* value of zero resets the counter to zero.

❖ *Note:* It is the sign of the increment, not the sign of the accumulated value of the `SpinCursor` counter, that determines the cursor's direction of spin.

Error message file management routines

Two procedures in the MPW Tools Library let you retrieve the text of error messages in the Macintosh Operating System error message file. These routines all use Pascal calling conventions and Pascal-formatted strings.

❖ *Note:* In the present version of MPW, the error message file manager accesses only the Macintosh Operating System error message file; future versions may include files specific to MPW and/or programs running under MPW.

To use the error message file manager, link your assembly with the file ToolLibs.o.

The InitErrMgr procedure

```
InitErrMgr(ToolErrFileName: Str255; SysErrFileName: Str255;
           ShowToolErrNbrs: boolean);

->      ToolErrFileName         name of your tool's error file, passed by pointer
->      SysErrFileName          name of the system error file, passed by pointer
->      ShowToolErrNbrs         1 to show tool error numbers, 0 otherwise
```

The `InitErrMgr` procedure must be called before any of the other error message file manager procedures. To access Macintosh Operating System error messages, push three longints with zero value on the stack and call `InitErrMgr`. This call causes the error manager to access the file SysErr.Err in the directory {ShellDirectory}, if that Shell variable is defined; otherwise it will use file SysErr.Err.

❖ *Note:* The type `Str255` is declared in the file MemTypes.p; it is equivalent to `STRING[255]`.

The GetSysErrText procedure

```
GetSysErrText(MsgNbr: integer; ErrMsg: StringPtr);

->      MsgNbr          the message number to be interpreted
<-      ErrMsg          address of the Pascal message string
```

The `GetSysErrText` procedure fetches the message text that corresponds to the system error number value of *MsgNbr*. *ErrMsg* is a pointer to a Pascal-formatted string of type `Str255`, in which the error message text will be placed.

If `GetSysErrText` is successful, the form of the error message returned is

error text (*OS error number*)

If it is unsuccessful, the form of the error message returned is

OS error number (*reason message not found*)

Possible reasons for unsuccessful execution of `GetSysErrText` are that the file SysErr.Err was not found or that it contained no message text corresponding to *MsgNbr*.

Appendix J

The Graf3D Library

Graf3D is a library that uses QuickDraw calls to produce three-dimensional graphics. It does this by providing a fixed-point interface to QuickDraw's integer coordinates.

The Graf3D routines provide several important features:

- A camera-eye view. This allows you to set the point of view from which the observer sees the object independently from the coordinates of the object itself. The camera is set up with the `ViewPort`, `LookAt`, and `ViewAngle` procedures. You can set the focal length of the camera as if you had a choice of telephoto, wide-angle, or normal lenses.
- Three-dimensional clipping to a true pyramid. The apex of the pyramid is at the point of the camera eye, and the base of the pyramid is equivalent to the `ViewPort`. When you use the Clip3D function, only objects upfront of the camera eye and within the pyramid are displayed on the screen.
- Two-dimensional point and line capability using Fixed coordinates. Graf3D provides commands corresponding to the QuickDraw commands but using `Fixed`-type coordinates instead of integers. With `Fixed` coordinates, you have a larger dynamic range for graphics calculations; with integer coordinates you get faster drawing time.
- Two-dimensional or three-dimensional rotation. You can rotate an object along any or all axes simultaneously, using the `Pitch`, `Yaw`, and `Roll` procedures.
- Translation and scaling of objects in one or more axes simultaneously. `Translation` means movement anywhere in three-dimensional space. `Scaling` means shrinking or expanding.

How to use Graf3D

To use Graf3D, do the following:

- Include the file Graf3DEqu.a in your source text.
- Link your assembly with the file {Libraries}Interface.o.
- Set values in the Graf3D data structures and call the Graf3D routines from your program, using the equates in Graf3DEqu.a.

In the rest of this appendix, Graf3D is described in MPW Pascal notation. For information on how to convert this notation into assembly-language calling conventions for stack-based routines, see the Using Assembly-Language chapter of *Inside Macintosh*.

Graf3D data types

Graf3D declares and uses these data types:

- Fixed
- Point3D
- Point2D
- XfMatrix
- Port3DPtr

The type `Fixed` is discussed in *Inside Macintosh*. The other types are discussed below.

Point3D

```
TYPE Point3D = RECORD
                 x: Fixed;
                 y: Fixed;
                 z: Fixed;
                 END;
```

Point3D contains three fixed-point number coordinates: x, y, and z. Graf3D uses x, y, and z for fixed-point number coordinates to distinguish between the h and v integer screen coordinates used by QuickDraw.

Point2D

```
TYPE Point2D = RECORD
                  x: Fixed;
                  y: Fixed;
                  END;
```

Point2D is just like Point3D but contains only x and y coordinates.

XfMatrix

```
XfMatrix = ARRAY[0..3,0..3] OF Fixed;
```

`XfMatrix` is a 4-by-4 matrix of fixed values, used to hold a transformation equation. Each transforming routine alters this matrix so that it contains the concatenated effects of all transformations applied.

Port3DPtr

```
Port3DPtr = ^Port3D;
Port3D    = RECORD
               grPort: GrafPtr;
               viewRect: Rect;
               xLeft,yTop,xRight,yBottom: Fixed;
               pen,penPrime,eye: Point3D;
               hSize,vSize: Fixed;
               hCenter,vCenter: Fixed;
               xCotan,yCotan: Fixed;
               ident: BOOLEAN;
               xForm: XfMatrix;
            END;
```

The type `Port3DPtr` contains all the state variables needed to map fixed-point number coordinates into integer screen coordinates. These are the variables:

Name	Description
grPort:	A pointer to the grafPort associated with this `Port3D`
viewRect:	The viewing rectangle within the grafPort; the base of the viewing pyramid
xLeft, yTop, xRight, yBottom	World coordinates corresponding to the `viewRect`
pen	A three-dimensional pen location

penPrime:	The pen location transformed by the `xForm` matrix
eye	A three-dimensional viewpoint location established by `ViewAngle`
hSize, vSize	Half-width and half-height of the `viewRect` in screen coordinates
hCenter, vCenter	The center of the `viewRect` in screen coordinates
xCotan, yCotan	The viewing cotangents set up by `ViewAngle`, used by Clip3D
ident	A `BOOLEAN` value that allows the transformation to be skipped when `xForm` is an identity matrix
xForm	A 4-by-4 matrix that holds the net result of all transformations

Graphics routines

Graf3D provides several procedures and functions to establish a graphics environment and create drawings within it. They are described in this section.

The InitGraf3D procedure

```
PROCEDURE InitGraf3D(globalPtr:  Ptr);
```

InitGraf3D starts up Graf3D and initializes its data structures. The `globalPtr` parameter is a pointer to heap space for Graf3D data.

Call `InitGraf3D` once and only once at the beginning of your program. Pass it the address of a `Port3DPtr` that you have reserved for use by Graf3D.

The Open3DPort procedure

```
PROCEDURE Open3DPort(port:  Port3DPtr);
```

The `Open3DPort` procedure initializes all the fields of a Port3D to their defaults, and makes that Port3D the current one. The pointer grPort is set to the currently open grafPort. These are the default values:

```
thePort3D := port;
port^.grPort := thePort;
ViewPort(thePort^.portRect);
WITH thePort^.portRect DO LookAt(left, top, right, bottom);
ViewAngle(X2Fix(0.0));
Identity;
MoveTo3D(X2Fix(0.0),X2Fix(0.0),X2Fix(0.0));
```

The SetPort3D procedure

PROCEDURE SetPort3D (port: Port3Dptr);

The `SetPort3D` procedure makes `port` the current Port3D and calls `SetPort` for that `Port3D`'s associated grafPort. `SetPort3D` allows an application to use more than one Port3D and switch between them.

TheGetPort3D procedure

PROCEDURE GetPort3D(VAR port: Port3DPtr);

The GetPort3D procedure returns a pointer to the current Port3D. This procedure is useful when you are using several Port3Ds and want to save and restore the current one.

The move procedures

Graf3D provides four different move procedures:

PROCEDURE MoveTo2D(x,y: Fixed);

PROCEDURE MoveTo3D(x,y,z: Fixed);

PROCEDURE Move2D(dx,dy: Fixed);

PROCEDURE Move3D(dx,dy,dz: Fixed);

These procedures move the pen in two or three dimensions without drawing lines. The fixed-point number coordinates are transformed by the `xForm` matrix and projected onto flat-screen coordinates; then Graf3D calls QuickDraw's `MoveTo` procedure with the result.

The Line Procedures

Graf3D provides four different Line procedures:

PROCEDURE LineTo2D(x,y: Fixed);

PROCEDURE LineTo3D(x,y,z: Fixed);

PROCEDURE Line2D(dx,dy: Fixed);

PROCEDURE Line3D(dx,dy,dz: Fixed);

These procedures draw two- and three-dimensional lines from the current pen location. The `LineTo2D` and `LineTo3D` procedures stay on the same *z*-plane. The fixed-point number coordinates are first transformed by the `xForm` matrix, then clipped to the viewing pyramid, then projected onto the flat-screen coordinates and drawn by calling QuickDraw's `LineTo` procedure.

The Clip3D Function

FUNCTION Clip3D(src1,src2: Point3D; VAR dst1,dst2: Point): BOOLEAN;

The `Clip3D` function clips a three-dimensional line segment to the viewing pyramid and returns the clipped line projected onto screen coordinates. `Clip3D` returns `true` if any part of the line is visible. If no part of the line is within the viewing pyramid, Clip3D returns `false`.

The set point procedures

PROCEDURE SetPt3D(VAR pt3D; Point3D; x,y,z: Fixed);

PROCEDURE SetPt2D(VAR Pt2D: Point2D); x,y: Fixed);

The `SetPt3D` procedure assigns three fixed-point numbers to a `Point3D`. The `SetPt2D` procedure assigns two fixed-point numbers to a `Point2D`.

Setting up the camera

The procedures `ViewPort`, `LookAt`, and `ViewAngle` position the image in the grafPort, aim the camera, and choose the lens focal length in order to map three-dimensional coordinates onto the flat screen space. These procedures may be called in any order.

The ViewPort procedure

PROCEDURE ViewPort(r: Rect);

The `ViewPort` procedure specifies where to put the image in the grafPort. The rectangle is in integer QuickDraw coordinates and tells where to map the `LookAt` coordinates.

The LookAt procedure

PROCEDURE LookAt(left,top,right,bottom: Fixed);

The `LookAt` procedure specifies the fixed-point number *x* and *y* coordinates corresponding to the `viewRect`.

The ViewAngle procedure

PROCEDURE ViewAngle(angle: Fixed);

The `ViewAngle` procedure controls the perspective by specifying the horizontal angle (in degrees) subtended by the viewing pyramid. Typical viewing angles are 0 degrees (no perspective), 10 degrees (telephoto lens), 25 degrees (the normal perspective of the human eye), and 80 degrees (wide-angle lens).

The transformation matrix

The transformation matrix allows you to impose a coordinate transformation between the coordinates you plot and the viewing coordinates. Each of the transformation procedures concatenates a cumulative transformation onto the `xForm` matrix. Subsequent lines drawn are first transformed by the `xForm` matrix, then projected onto the screen as specified by `ViewPort`, `LookAt`, and `ViewAngle`.

The Identity procedure

PROCEDURE Identity;

The `Identity` procedure resets the transformation matrix to an identity matrix.

The Scale procedure

PROCEDURE Scale(xFactor,yFactor,zFactor: Fixed);

The `Scale` procedure modifies the transformation matrix so as to shrink or expand by `xFactor`, `yFactor`, and `zFactor`. For example,

```
Scale(X2Fix(2.0),X2Fix(2.0),X2Fix(2.0))
```

will make everything come out twice as big when you draw.

The Translate procedure

PROCEDURE Translate(dx,dy,dz: Fixed);

The `Translate` procedure modifies the trasnsformation matrix so as to displace by `dx, dy, dz`.

The Pitch procedure

PROCEDURE Pitch(xAngle: Fixed);

The `Pitch` procedure modifies the transformation matrix so as to rotate `xAngle` degrees around the *x* axis. A positive-Angle rotates clockwise when looking at the origin from positive *x*.

The Yaw procedure

PROCEDURE Yaw (yAngle: Fixed);

The `Yaw` procedure modifies the transformation matrix so as to rotate `yAngle` degrees around the y axis. A positive-Angle rotates clockwise when looking at the origin from positive *y*.

The Roll Procedure

PROCEDURE Roll(zAngle: Fixed);

The `Roll` procedure modifies the transformation matrix so as to rotate `zAngle` degrees around the *z* axis. A positive-Angle rotates clockwise when looking at the origin from positive *z*.

The Skew procedure

PROCEDURE Skew(zAngle: Fixed);

The `Skew` procedure modifies the transformation matrix so as to skew `zAngle` degrees around the *z* axis. Skew changes only the *x* coordinate; the result is much like the slant QuickDraw gives to italic characters. For instance, `Skew(X2Fix(15.0))` makes a reasonable italic. A positive-Angle rotates clockwise when looking at the origin from positive *z*.

The TransForm Procedure

PROCEDURE TransForm(src: Point3D; VAR dst: Point3D);

The `Transform` procedure applies the `xForm` matrix to `src` and returns the result as `dst`. If the transformation matrix is `Identity`, `dst` will be the same as `src`.

Appendix K

Object Assembler Macros

The file ObjMacros.a contains a collection of MPW assembly-language macros. The macros permit you to write applications, or parts of applications, in an object-oriented fashion using assembly language.

Object-oriented assembly language allows you to

- create descendants of objects defined in Object Pascal
- call methods written in Object Pascal from assembly-language code
- call methods written in assembly language from Object Pascal

Object-oriented assembly language is as much a style of programming as a language. This chapter describes the macros provided for use with assembly language. For more complete information about object-oriented programming in general, see the *Macintosh Programmer's Workshop Pascal Reference.* For a sample program that uses object-oriented macros, see the *MacApp User's Guide.*

InitObjects

InitObjects is a macro that must be called at the beginning of every program written entirely in assembly language using Objects. In other words, this must be the first line of every object-oriented assembly-language program that isn't linked with an Object Pascal program.

InitObjects Prototype Statement

```
InitObjects
```

Example

```
InitObjects
```

ObjectDef

The `ObjectDef` macro allows you to define objects in object-oriented assembly-language programs. The `ObjectDef` macro also generates code so method calls are handled properly.

The `ObjectDef` macro sets up Assembler record definitions so that you can later refer to fields of objects by using `%Typename` and the field name. For example, if an object has this definition:

```
ObjectDef   TArc,TShape,              \
            (nextShape, TShape),      \
            (arcAngle,2),             \
            METHODS,                  \
            (Draw)
```

you can use a statement like this:

```
MOVE.W      %TArc.arcAngle(A0),-(SP).
```

A percent sign (%) is appended to the object-type identifier (other than when used in the `ObjectDef` and `ObjectWith` macros) so that you can use the Object Pascal object-type identifier in an `EQU` statement to define the size of an object and then use that name in place of the size for a field that refers to an object of that type. For example

```
TShape      EQU         4
```

was used to set the value of the identifier `TShape` for the above `ObjectDef`.

❖ *Note:* Because an object reference is a handle, the `EQU` value will always be 4.

In object-oriented assembly language you must qualify the field name by the name of the object type that defined it. For example, if `TArc` inherits the field `fColor` from TShape, %TArc.fColor would be undefined. You would have to call it %TShape.fColor. The ObjectWith macro (described below) is provided to get around this problem.

ObjectDef prototype statement

```
ObjectDef     &TypeName,&Heritage[,fieldList][,METHODS,methodList]
```

❖ *Note:* The part of the prototype statement for ObjectDef that is in square brackets is optional. *FieldList* should be replaced by a list of all fields of the object type, with a size for each field, all in parentheses, as shown in the examples. *MethodList*, similarly, should be replaced by a list of the methods of the object type. If you have a method list, the word `METHODS` must be present. Otherwise, it should never be present.

Examples

```
ObjectDef Shape, TObject,           \
            (boundRect,8),          \
            (borderThickness,2),    \
            (color,2),              \
            METHODS,                \
            (Draw),                 \
            (MoveBy),               \
            (Stretch)

ObjectDef Arc,Shape,                \
            (startAngle,2),         \
            (arcAngle,2),           \
            METHODS,                \
            (Draw,OVERRIDE),        \
            (GetArea),              \
            (SetArcAngle)
```

The numbers given after the field identifiers give the size, in bytes, of the storage required.

❖ *Note:* The backslash (\) is required by the MPW Assembler when you continue a statement from one line to the next.

If a method is inherited and reimplemented, you must qualify the method name with the word `OVERRIDE` as shown for the Draw method.

ObjectIntf and the IMPL keyword

`ObjectIntf` is used to create an interface in assembly language for an object type that is declared in Object Pascal. It is the same as `ObjectDef` except that it does not generate method tables.

This macro is generally used when you want an assembly-language implementation of a method declared in an Object Pascal unit. It allows you to specify which methods will be implemented in assembly language. In the Object Pascal unit, you should declare the method `EXTERNAL`. For example:

```
TFoo = OBJECT(TObject)
         field1:  INTEGER;
         PROCEDURE TFoo.Meth1;
         PROCEDURE TFoo.Meth2;
       END;

IMPLEMENTATION
  PROCEDURE TFoo.Meth1; EXTERNAL;
  PROCEDURE TFoo.Meth2;

BEGIN
- - -
END;
```

In the Assembler file, you need to supply an `ObjectIntf` template for the class. You must give the entire object-type declaration in the `ObjectIntf` template. You give the `IMPL` keyword, preceded by a comma, after any method you want to implement in Assembler. The "Examples" subsection below give the corresponding assembly-language declaration for the Object Pascal declaration given above.

If the method for which you are providing an Assembler implementation is an override of an inherited method, you must specify both `OVERRIDE` and `IMPL`:

```
            ObjectInf   MyShape, Shape          \
                        - - -                   \
                        (Draw, OVERRIDE, IMPL), \
                        - - -                   \
```

ObjectIntf prototype statement

ObjectIntf &TypeName,&Heritage[,*fieldList*][,METHODS,*methodList*]

The part of the prototype statement for `ObjectIntf` that is in square brackets is optional. *FieldList* should be replaced by a list of all fields of the object type, with a size for each field as shown in the examples. *MethodList*, similarly, should be replaced by a list of the methods of the object type. If you have a method list, the word `METHODS` must be present. Otherwise, it should never be present.

Examples

```
            ObjectIntf  TFoo, TObject,          \
                        (field1, 2),            \
                        METHODS,                \
                        (Meth1, IMPL),          \
                        (Meth2)

Meth1       ProcMethOf  TFoo

            EndMethod
```

; The following creates an assembly-language interface for TObject

```
ObjectInf   TObject,,          \
            METHODS,           \
            (Free),            \
            (ShallowFree),     \
            (Clone),           \
            (ShallowClone)
```

ObjectWith and EndObjectWith

The `ObjectWith` and `EndObjectWith` macros allow you to specify a field of an object without having to qualify it with the object reference.

As noted before, you must normally qualify the name of an inherited field with the name of the object type that defined it. These macros allow you to avoid that. `ObjectWith` inserts a series of MPW assembly-language `WITH` directives, one for each ancestor class, so you can specify fields of any ancestor object type without qualifying it with the ancestor object type name. `ObjectWith` can be nested. The most recent invocation has precedence when there are field name conflicts. You end an `ObjectWith` block with an `EndObjectWith`.

ObjectWith and EndObjectWith prototype statements

```
ObjectWith          &TypeName
EndObjectWith
```

Examples

```
ObjectWith          Shape
MOVE.W              color(A1),D0
EndObjectWith

ObjectWith          Arc
MOVE.W              startAngle(A1),-(SP)
PEA                 boundRect(A1)
EndObjectWith
```

ProcMethOf, FuncMethOf, and EndMethod

These macros are used to bracket methods. `ProcMethOf` and `FuncMethOf` invoke the `PROC` and `FUNC` directives. They provide the implicit parameter `SELF`, which is a reference to the object used to call the method. They also invoke the `ObjectWith` macro with the specified type name so that fields of `SELF` can be accessed without type-name qualification. `EndMethod` invokes the `EndObjectWith` macro and the `ENDPROC` directive.

As with any assembly-language routine, before ending the routine (that is, before the EndMethod macro) you must remove the parameters from the stack. In a method, you should be careful to also remove the implicit parameter SELF. See the sample program at the end of this appendix for examples of this.

ProcMethOf, FuncMethOf, and EndMethod prototype statements

```
&ProcName   ProcMethOf          &TypeName

&ProcName   FuncMethOf          &TypeName

            EndMethod
```

Examples

```
Draw        ProcMethOf          Shape
            ---
            EndMethod

            GetArea             FuncMethOf Arc
            ---
            EndMethod
```

MethCall

The MethCall macro is used to invoke methods. If the second parameter is omitted, the current method's object type is assumed. The MethCall macro generates a JSR to the proper method.

MethCall prototype statement

```
            MethCall            &ProcName,&TypeName
```

Examples

```
            MOVE.L              A2,-(SP)
            MethCall            Draw
            MOVE.W              D0,-(SP)
            MOVE.W              D1,-(SP)
            MOVE.L              aShape(A6),-(SP)
            MethCall            MoveBy,Shape
```

Inherited

The `Inherited` macro calls the method of the given name in the closest ancestor object type that implemented that method.

Inherited prototype statement

```
Inherited       &ProcName
```

Example

```
MoveSelf        -(SP)
Inherited       Draw
```

NewObject

The `NewObject` macro, as its name suggests, is used to create a new object. It is equivalent to the Pascal `New` procedure used with an object reference.

NewObject generates a `JSR` to `%_OBNEW` after pushing the appropriate parameters onto the stack. If `&Size` is omitted (and it usually is) the instance size for the given object type is used. The `&Loc` parameter must be a memory reference.

NewObject prototype statement

```
NewObject       &Loc,&TypeName,&Size
```

Examples

```
NewObject       -4(A6),Arc
NewObject       gArray,DynArray,200
```

MoveSelf

This `MoveSelf` macro is a convenience macro. It executes the following statement

```
MOVE.L          8(A6),&Dest
```

`MoveSelf` assumes that the method began with a `LINK A6,#nnnn`. `(8(A6)` is the location of `SELF` when in a method.)

MoveSelf prototype statement

```
MoveSelf        &Dest
```

Examples

```
MoveSelf        A4
MoveSelf        -(SP)
MoveSelf        aSquare(A6)
```

Appendix L

Pascal and C Calling Conventions

This appendix describes the conventions used in Pascal and C to pass parameters, return function results, and save and restore register contents during procedure and function calls.

Pascal calling conventions

This section covers parameter passing, function returns, and register conventions in Pascal.

Parameters

Pascal parameters are evaluated from left to right and are pushed onto the stack in that order as they are evaluated. The called procedure is responsible for removing the parameters from the stack. All VAR parameters are passed as pointers to the actual storage location. In cases of byte-wide types, VAR parameters may have odd absolute values.

Non-VAR parameters are passed in different ways, depending on the type of the parameter. Values of type `Boolean`, elements of an enumerated type with fewer than 128 elements, and subranges within the range –128..127 are passed as signed byte values. (They are pushed as bytes; the 68000 allocates two bytes for each byte on the stack.) The called procedure expects `boolean` parameters to be in the range 0..1. Values of types `Integer`, `char`, and all other enumerations and subranges are passed as signed word values. Pascal `char` values are expected to be in the range 0..255; the upper half of this range is used for special characters. Pointers and `longint` values are passed as signed 32-bit values.

Real-type parameters

Values of types `real`, `double`, `comp`, and `extended` are passed as pointers to extended values. The Compiler does this in a reentrant way by allocating a temporary location in the caller's activation record, converting the parameter value to an `extended` value in this location, and passing a pointer to this location. The called procedure then allocates a local location of the declared type and converts the `extended` value, using the pointer, into the location and type.

Structured-type parameters

Arrays, strings, and records whose size is less than or equal to four bytes are passed by pushing their value (either a word or a long word) onto the stack. Larger arrays, strings, and records (as well as `extended` values, as mentioned above) are passed as a pointer to the value; for reentry purposes, the Compiler emits code in the called procedure to copy the value to a local storage location.

Sets are passed by rounding the set size up to the next whole word, if necessary, then pushing the set value so that the lowest-order word is pushed last. In the case of a byte-width set, the called procedure will only access the low-order half of the word pushed.

Table L-1
Parameter passing conventions

Parameter type	Pascal caller...	Pascal callee...
`boolean`	Pushes byte: range 0..1	Accesses byte: range 0..1
enumeration: range 0..127	Pushes byte: range 0..127	Accesses byte: range 0..127
enumeration: range 0..32767	Pushes word: range 0..32767	Accesses word: range 0..32767
`char`	Pushes word: range 0..255	Accesses word: range 0..255
subrange: range –128..127	Pushes byte: range –128..127	Accesses byte: range –128..127
subrange: range –32767..32767	Pushes word: range –32767..32767	Accesses word: range –32767..32767
`integer`	Pushes word: range –32767..32767	Accesses word: range –32767..32767
`longint`	Pushes long	Accesses signed long value
pointer	Pushes long	Accesses long
`real`	Converts to `extended`, pushes address of `extended`	Converts `extended` on stack to local `real`, accesses local value
`double`	Converts to `extended`, pushes address of `extended`	Converts `extended` on stack to local `double`, accesses local value
`comp`	Converts to `extended`, pushes address of `extended`	Converts `extended` on stack to local `comp`, accesses local value
`extended`	Pushes address of `extended`	Copies `extended` to local `extended`, accesses local value
`ARRAY`, `RECORD`, `STRING` ≤ four bytes	Pushes value (word or long)	Accesses value (word or long)
`ARRAY`, `RECORD`, `STRING` > four bytes	Pushes address of value	Copies value to local, accesses local
`SET`	Pushes set value rounded to whole number of words	Accesses value on stack (*Note:* Use word or long for sets of those sizes; accesses low-order half of word for byte-size set.)

Function results

Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the parameters are pushed. The caller is responsible for removing the result from the stack after the call.

For types `boolean`, `char`, `integer`, and enumerated and subrange types, the caller allocates a word on the stack to make space for the function result. Values of type `Booleans`, enumerated types with fewer than 128 elements, and subranges within the range –128..127 are returned as signed byte values. The value goes in the low-address byte, which is the most significant byte of the word. The calling procedure expects `boolean` results to be in the range 0..1.

`Integer` and `char` values and all enumerated and subrange types not covered above are returned as signed word values. Pascal `char` values are expected to be in the range 0..255; the upper half of this range is used for special characters.

For pointer, `longint`, and the real types, the caller allocates a long word on the stack to make space for the function result. Pointers and `longint` values are returned as signed 32-bit values. Values of type `real` are returned as 32-bit real values. For `double`, `comp`, and `extended` types, and also for sets, arrays, strings, and records greater than four bytes in size, the caller pushes a pointer to a temporary location.

For one-byte sets and for arrays, strings, and records whose size is one word, the caller allocates a word on the stack. For sets, arrays, strings, and records whose size is two words, the caller allocates a long word on the stack. One-byte sets are returned as a byte value. Sets, arrays, strings, and records whose sizes are one or two words are returned as either a word or a long word.

Table L-2
Function-result passing conventions

Result type	Pascal Caller action	Pascal Callee action	After the call
`boolean`	Allocate word	Return byte value: range 0..1	Pop byte
Enumeration: range 0..127	Allocate word	Return byte value: range 0..127	Pop byte
Enumeration: range 0..32767	Allocate word	Return word value:	Pop word
`char`	Allocate word	Return word value: range 0..255	Pop word
Subrange: range –128..127	Allocate word	Return byte value: range –128..127	Pop byte

Subrange: range –32768..32767	Allocate word	Return word value: range –32768..32767	Pop word
`integer`	Allocate word	Return word value: range –32768..32767	Pop word
`longint`	Allocate long word	Return long word value range: signed 32 bits	Pop long word
`real`	Allocate long word	Return `real` value	Pop real value
`double`	Push address of `double` temporary	Put `double` result in temporary	Pop temporary address access temp value
`comp`	Push address of `comp` temporary	Put `double` result in temporary	Pop temporary address access temp value
`extended`	Push address of `extended` temporary	Put `extended` result in temporary	Pop temporary address access temporary value
ARRAY STRING RECORD ≤ four bytes	Allocate word or long word	Return word or long	pop word or long
ARRAY STRING RECORD > four bytes	Push address of temporary	Put result in temp	Pop temporary address access temporary value
SET one byte	Allocate word	Return byte value of result	Pop byte
SET one word	Allocate word	Return word value of result	Pop word
SET two words	Allocate long word	Return long word value of result	Pop long
SET > two words	Push address of temp	Put result in temp	Pop temporary address access temporary value

Register conventions

Registers D0, D1, D2, A0, and A1 are considered scratch registers and are not preserved across procedure calls. All other registers are preserved by the called routine. Register A5 is the global frame pointer, register A6 the local frame pointer, and register A7 the stack pointer.

C calling conventions

This section covers the treatment of parameters, function results, and registers in C.

Parameters

Parameters to C functions are evaluated from right to left and are puched onto the stack in the order they are evaluted. Characters, integers, and enumerated types are passed as sign-extended 32-bit values. Ponters and arrays are passed as 32-bit addresses. Type `float`, `double`, `comp`, and enteded are passed as extended 80-bit values. Structures are also passed on the stack. Their value is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the nightes memory address. The caller removes the parametes from the stack.

Function results

Characters, integers, enumerated types, and ponters are returned as sign-extended 32-bit values in register D0. Types float, double, comp, and extended are returned as extended values in registers D0, D1, and A0. The low-order 16 bits of D0 contain the sign and exponent bits; register D1 contains the high-order 32 bits of the significand; register A0 contains the low-order 32 bits of the significand. Stucture values are returned as a 32-bit ponted in registe D0. the pointer contains the address of a static variable into which the result is copied before returning. This implementaion of stucture function results is not reentrant.

Register conventions

Registers D0, D1, A0, and A1 are scratch registers that are not preserved by C functions. All other registers are preserved. Register A5 is the global frame pointer, register A6 is the local frame pointer, and register A7 is the stack pointer. Local stack frames are not necessarily created for simple functions.

Appendix M

The MC68030 Processor

The MC68030 processor is a single chip that combines most of the capabilities of the MC68020 processor with some (but not all) of the capabilities of the MC68851 Paged Memory Management Unit coprocessor. This appendix tells you how to write source text for the MC68030 processor.

Assembler control

Source text written for the MC68030 processor must contain the following directive before any MC68030 operations:

[*macro-label*] `MACHINE` `MC68030`

This tells the Assembler to process all subsequent code to run on the MC68030.

The source text must not contain an `MC68851` directive as well; if it does, the Assembler will report an error. However, it may include an `MC68881` directive.

After a `MACHINE MC68030` directive, the `&SETTING` function will return a value of `MC68030` when its operand is `MACHINE`. The `&SETTING` function is described in Chapter 5.

MC68020 statements you can use

In source text intended for the MC68030 processor, you may use any of the instructions and directives valid for the MC68020 except the `CALLM` and `RTM` instructions. These may be used only with the MC68020 processor.

MC68851 instructions you can use

The MC68030 processor contains only four of the MC68851 registers (in addition to the MC68020 registers), and can execute only some of the MC68851 coprocessor instructions. The registers are listed in Table M-1; the valid coprocessor instructions are listed in Table M-2.

Table M-1
MC68851 registers in the MC68030

Designation	Usage
CRP	CPU root pointer register
SRP	Supervisor root pointer register
PSR	PMMU status register
TC	Translation control register

Table M-2
MC68851 instructions valid for the MC68030

Opcode	Operand format	Sizes	Notes
PFLUSH	*fc,#ae[,ea]*		1
PFLUSHA			
PLOADR	*fc,ea*		1
PLOADW	*fc,ea*		1
PMOVE	*PMMU-reg,ea*	B | W | L | D	2
PMOVE	*ea,PMMU-reg*	B | W | L | D	2,3
PTESTR	*fc,ea,#ae[,An]*		1
PTESTW	*fc,ea,#ae[,An]*		1

1. The function code is defined as follows:

   fc ::= *#ae* (specified as four bits in the command word)
   Dn (contained in the lower four bits of D*n*)
   SFC (contained in the processor's source function register)
   DFC (contained in the processor's destination function code register)

2. The PMMU registers are listed in Table M-1.

3. The root pointer registers in the MC68030 contain double long words, 64 bits long. The PMOVE instruction, which references these registers, accepts immediate effective addresses; hence for *ea* you can use *#ae* (mode 74, Table 3-2). If you do this, however, the Assembler will convert the 32-bit effective address to a 64-bit value by filling it on the left with 32 zero bits. It will also issue a warning, because it does not support 64-bit values. You can avoid this limitation by defining a constant with two DC.L directives, then referencing them in your PMOVE instruction.

Glossary

@-label: A label of very limited scope, written with @ as its first character.

absolute: Of an expression, having a value that can be determined during assembly.

actual parameter: A parameter in a *macro call.*

addressing mode: see **effective addressing mode**.

anonymous: Of a file or variable, not having an identifier. Anonymous objects are accessed by pointers.

application globals area: An area in RAM in which application programs store data.

application parameter area: An area in RAM pointed to by register A5.

array: A data structure containing an ordered set of elements.

ASCII: Acronym for American Standard Code for Information Interchange; a system of assigning code numbers to letters, numerals, punctuation marks, and control codes.

ASCII character: A character whose ASCII code number lies in the range 0..127.

as-is string: A string that the Assembler stores without length information.

assembler: A program that translates *source text* into *object code.*

Assembler option: An instruction passed to the Assembler at the time it is invoked.

Assembler system variable: A variable whose value is determined by the Assembler.

assembly: The process of translating *source text* into *object code.*

assembly-control directive: A directive that controls whether or not some portion of source text is assembled.

backquote: The ASCII $60 character, written `, which tells the Macro Processor that the next character is to be processed literally during macro expansion.

binary: Base-2 number representation, using the numerals 0 and 1.

blank: A tab or space character.

body: In a macro definition, the machine instruction and directive statements that comprise the macro, other than the MACRO directive, prototype line, and ENDM directive.

Boolean expression: An expression whose value is either `true` or `false`.

built-in function: A function that is part of the macro language.

C string: A string in a format compatible with the C programming language; it is terminated with a 0 byte.

code: Executable computer instructions.

command line: The text you enter to execute a command in the MPW Shell.

comment: Source text intended for a human reader, ignored by thew Assembler.

comment field: The area of a source text line reserved for comments.

comment line: A source text line containing only a comment.

comparison operator: An operator that compares two values; they are listed in

Table 2-3.

conditional assembly: The process of controlling what source text is assembled.

continuation character: A character at the end of a source text line that lets you continue the text onto a second line. Backslash (\) is the Assembler's continuation character.

data: Information that a computer processes.

decimal: Base-10 number representation, using the numerals 0 -9.

destination: An address into which an instruction places data.

diagnostic output file: A file to which MPW tools, including the Assembler, write error messages and progress information.

dimension: An ordering relation among elements of an *array.*

directive statement: A source text instruction to the Assembler, which generates no direct code.

dynamic nesting level: The ordinal position of a macro call in a *macro call chain.*

element: One item in a *sublist.*

effective addressing mode: Any of several ways of writing an address so it can be assembled.

equal: (Of strings) indistinguishable.

equate: An `EQU` or `SET` directive.

error code: A code (usually a number) returned by the Assembler to indicate that it has encountered an error in the source text or assembly process.

expand: To replace a macro call with the appropriate macro body, substituting actual values for variables and parameters where needed.

export: To make a module or entry point defined in the current assembly linkable to other assemblies.

field: A data structure within a *template* or *statement.*

first-level call: The *outer macro call* of a *macro call chain.*

floating-point: A way of representing decimal numbers.

formal parameter: A parameter in a *macro* definition.

generic form: A way of writing a statement so that the Assembler will convert it into a different form.

global scope: The scope of code or data that is accessible in more than one module.

global symbol table: A symbol table that contains symbols with global scope.

header: In a *macro* definition, the `MACRO` directive itself.

hexadecimal: Base-16 number representation, using the numerals 0 -9 and the letters A -F.

identifier: A name in source text.

IF...ENDIF construct: Source text enclosed by an `IF` directive followed by an `ENDIF` directive.

import: To make a module or entry point defined in another assembly linkable to the current assembly.

include: To insert the contents of a source text file into an *assembly.*

index: A numeric value that indicates the position of an element in a *sublist* or *array*, expressed by a subscript.

initial value: The value the Macro Processor assigns to a SET variable when it is created.

inner macro call: A macro call inside a macro.

instruction: A program element representing a single computer operation.

jump table: In the Macintosh, a table of references in RAM, used for communication between segments.

keyword: An identifier for a macro parameter.

keyword macro: A macro whose parameters are identified by keywords.

keyword macro call: A call to expand a keyword macro.

label: A name that identifies a location in assembled code or data.

label field: The leftmost field of a *statement*.

lifetime: Of a variable, the duration of program execution during which it is accessible.

literal: Immediate data given to an instruction.

literalize: To modify a symbol in source text so that the Assembler does not interpret it.

literal pool: A table of all literals in an assembly, without duplications, maintained by the Assembler.

local scope: The scope of code or data that is accessible in only one module.

local symbol table: A symbol table that contains symbols with local scope.

location counter: A counter maintained by the Assembler that points to the current byte of code or data being assembled.

logical operator: One of the operators OR, XOR, AND or NOT.

machine instruction statement: A statement that generates executable code.

macro: Defined source text that the Macro Processor expands into other source text on command.

macro call: A command to the Macro Processor to insert a macro at a specific point in a source text.

macro call chain: A sequence of one *outer macro call* and any number of *inner macro calls*.

macro definition: The source text that constitutes a macro.

macro directive: A directive that controls macro expansion but does not generate any source text directly.

macro label: A label indicating a location in source text, used only with GOTO directives.

macro language: The directives and functions that create and manipulate macros.

Macro Processor: The part of the Assembler that interprets the macro language.

macro symbol table: A symbol table that contains macro symbols and definitions.

macro variable: A variable whose value is controlled by macro directives.

main code module: The code module that will be executed first when a program runs.

main data module: A specific data module that the Linker places at the base of the global data area, where its offset is known to the Assembler.

mixed-mode macro: A macro containing both *positional* and *keyword parameters*.

mnemonic: A source text symbol that expresses an *instruction* or *directive*.

model statement: In a macro body, a statement that the Macro Processor uses as a model to generate actual machine instruction or directive statements.

module: A contiguous collection of *code* or *data*.

module directive: A directive that defines a code or data module.

non terminal symbol: Part of a syntax diagram that stands for something to be written in the source text, such as *expr* for an expression.

numeric constant: A number expressed directly in a source text.

object code: Machine-language code generated by the Assembler.

object file: A file containing specifications for data and code module contents, references to other code and data modules, and segmentation information.

operand field: The source text column that contains instruction operands and directive parameters.

Operating System routine: A Macintosh library routine that performs a task such as accessing a disk file or handling an error.

operation: A *directive* or *instruction* in a source text.

outer macro call: A macro call that is not inside a macro.

paired single quotation marks: The first and last single quotation marks in a *quoted string*.

paired brackets: A left bracket and a right bracket that the Assembler treats as enclosing an expression.

paired parentheses: A left parenthesis and a right parenthesis that the Assembler treats as enclosing an expression.

parameter: A piece of data in the operand field of a directive, macro prototype, or macro call statement.

parameter list: A sequence of parameters, separated by commas, in a macro prototype or macro call.

Pascal string: A string assembled into a form compatible with the Pascal programming language; it begins with a length byte and has a maximimum length of 255 characters.

pass: One processing cycle of the Assembler.

PC-relative: Located relative to the current value of the program counter.

positional macro: A macro whose parameters are identified by their position in its parameter list.

positional parameter: A macro parameter identified by its position in the parameter list.

postfix notation: A way of representing *object code* in an incomplete form, used by the Assembler on its first pass.

predefined SET variable: A SET variable whose value is set by the *Macro Processor*.

prototype: In a *macro definition*, the statement that establishes the name and parameter format of the macro.

qualified: Said of an identifier that is written with a qualifier.

qualifier: An identifier appended to another identifier(usually with a period between), which modifies its meaning.

quoted string: A string enclosed in single quotation marks (').

register list: A source text list of microprocessor registers, used as an operand of move-multiple instructions.

relative ASCII ordering: The algorithm by which the Macro Processor compares strings.

relocatable: Of an expression, having a value that cannot be determined during assembly.

Macintosh library routine: Any of the standard Macintosh routines described in *Inside Macintosh*.

scope: The area of source text in which a piece of **code** or **data** can be referenced.

scratch file: An area of RAM or disk memory used for temporary storage.

second-level call: The first *inner macro* call in a **macro call chain.**

segment: A collection of modules that is loaded together from disk into RAM during program execution.

SET variable: A macro variable whose value is assigned by a SETA or SETC directive.

source: In *instruction* syntax, an address from which the instruction takes data.

source text: Program text written by a programmer.

statement: A line of *source text,* including machine instruction statements and directive statements.

string: A sequence of one or more characters.

string constant: A string written explicitly in source text, enclosed in single quotation marks.

subscript: A numeric expression whose value is the index of an element in a sublist or array.

sublist: A collection of macro call parameters that the Macro Processor treats as a unit.

symbol: A lexical component of source text processed by the Assembler.

symbolic parameter: A variable that acquires a value during macro expansion.

symbol table: A list of source text symbols and their values maintained by the Assembler.

table: An ordered list of code or data objects that the Assembler creates in memory during assembly.

template: A source text structure that describes a collection of data without allocating memory for it.

terminal symbol: Part of a syntax diagram that must be written in the source text exactly as it appears in the diagram.

tool: A program that runs in the MPW Shell environment.

Toolbox routine: A Macintosh library routine that performs a task such as creating a menu or manipulating a window.

token: A character or group of characters that the Assembler interprets as a single syntactic entity.

trailer: In a macro definition, the ENDM or MEND directive.

trap dispatcher: A routine in RAM that handles unimplemented instructions, among other tasks.

unequal: Of strings able to be distinguished.

unimplemented: Of an *instruction,* one that is not part of the standard MC68xxx instruction set.

variable definition directive: A directive that creates a macro variable.

WHILE...ENDWHILE construct: Source text enclosed by a WHILE directive followed by an ENDWHILE directive.

MPW & MacApp Bug Report Form

BACKGROUND

Date _____ Version _____

AREA: Compiler: C Pascal

Assembler

Library: C Pascal Assembly

MacApp

Shell/Editor

Tool _____

Performance

BUG DESCRIPTION

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

CONTACT INFORMATION

Name: _____ Phone/Ext. _____

Address: _____

City, State, Zip _____

Please return completed form to:
MPW Bug Report; MS 27S; Apple Computer, Inc.; 20525 Mariani Ave.; Cupertino, CA 95014